阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 阿里巴巴集团 F2E | D2 前端技术论坛

2022 前端技术趋势解读

带你把握最新前端技术趋势



作　　者： D2前端技术论坛

推荐语： 阿里巴巴前端委员会推荐！
五大领域技术实战及多样化技术趋势分享，不容错过。

VISIT…

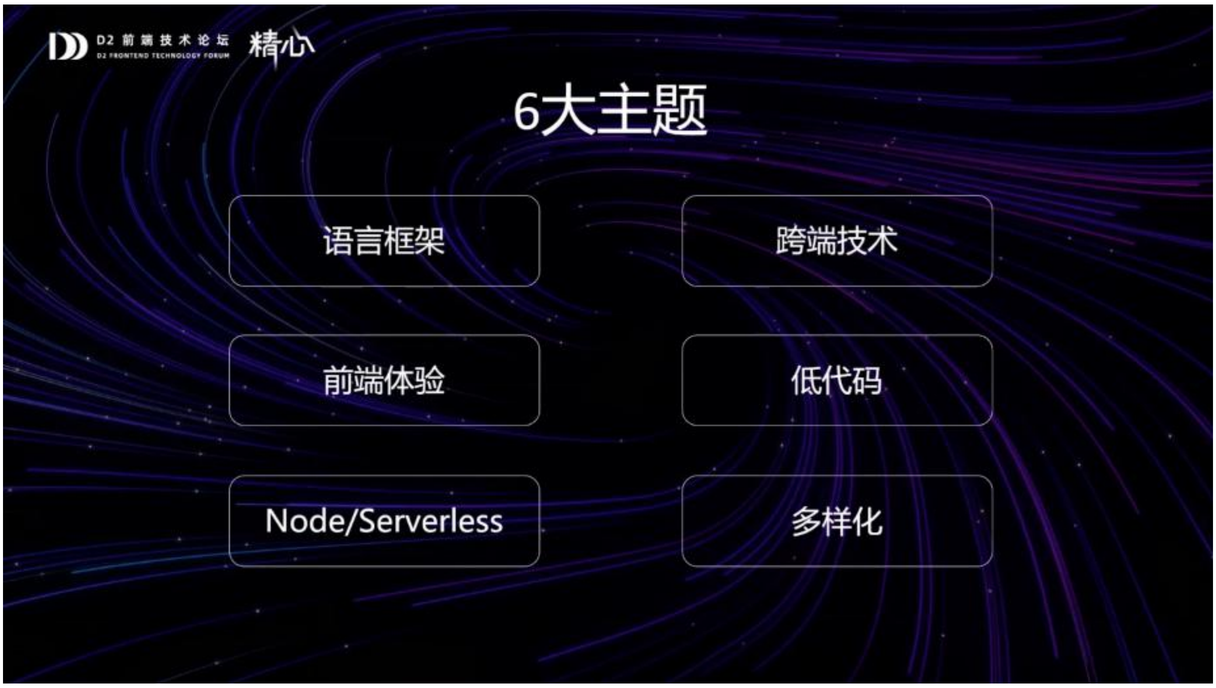
LANZAROTE Caliente.com

序

演讲人：姜凡(展炎)，阿里巴巴前端委员会委员长，大淘宝前端团队负责人，推进前端工程化、端架构演进来不断提升用户体验，致力通过 Node、智能化、产品平台能力建设，演进&定义“新前端”。

正文：大家好，我是阿里巴巴前端委员会委员长姜凡，花名展炎。首先感谢大家一直以来对 D2 前端技术论坛的关注和支持。

今年第十六届 D2 前端技术论坛的 slogan 是“精心”，所谓“精”是指对体验的精益求精，“心”则是指回归用户体验的初心。根据这个 slogan，本次大会组委会拟定了六个主题方向，包括：语言框架、跨端技术、前端体验、Node/Serverless、低代码、多样化。



下面，我分别引用一句话，来描述这六个方向。

1、语言框架

“形而上者谓之道，形而下者谓之器”——《易经·系辞》

语言框架是 D2 一直保留的专场，有些人可能觉得它有些生涩或实用性不高，而随着技术体系和生态的发展，前端是技术领域里生态最繁华的，究其原因是跟前端语言框架的发展和形态有很大关系。

我们希望今天在具体的研发、工程框架、解决方案之外，能够站在更高的视角来看待其背后的演进和发展思路到底是什么，这也是我们一直以来保留语言框架这个专场的原因。

2、跨端技术

“不可陷之楯与无不陷之矛,不可同世而立”——《韩非子》

今天在各个端之间，各个厂商之间，都希望建立起一个高墙，就像一个盾一样把自己的商业和生态圈起来；而从研发角度则希望能够敏捷、快速试错、低成本，就像矛一样去打破这个壁垒。

其实我们知道，不可能有一个盾把某个商业完全隔离，也不可能有一个方案能够把这些高墙全部击碎，二者之间就是不断在寻找一个动态的平衡的过程，来解决商业壁垒或商业鸿沟与通用性之间的平衡。

3、前端体验

“不忘初心，方得始终”——《华严经》

前端体验专场是今年独立设定的，这也是今年 D2 论坛的 slogan“精心”的缘起之一，重新声明了用户体验再次回归的价值。

4、Node/Serverless

“行而不辍，未来可期”——《荀子·修身》

Node/Serverless 专场在 D2 也是延续了很长时间，我们对它的发展投入了大量工作。因为 Node 的历史并不长，存在比如多语言的问题，比如原来仅仅是基于高并发、而今天要回归到弹性扩缩容的问题，以及并不是那么完备的线上运维的问题等等。

客观来讲，相比虚拟机和容器的市场份额，Serverless 的市场份额并不是很大。但无论是从技术研发角度还是从云原生厂商角度，都非常看好 Serverless 的前景和市场价值。

5、低代码

“旧时王谢堂前燕，飞入寻常百姓家”——《乌衣巷》

这段话的原意是表达的旧贵族的落寞，但我在这里想表达的是：低代码除了我们今天关注的效能和成本之外，它一方面解决了在不懂技术或没有技术资源的情况下，有一个好的 idea 就可以通过低代码实现，就像我们今天使用电脑的办公软件一样；另一方面，低代码降低了很多门槛，让更多的人有机会进入到社会数字化大升级的战役当中来。

换句话说，过去只有少部分人可以做的事情，今天低代码让更多的人可以使用，让他们的产出变得更加有价值，这是一个“普惠”的思想。

6、多样化

“乱花渐欲迷人眼，浅草才能没马蹄”——钱塘湖春行

今天的多样化非常繁华，其实多样化的场景完全是因为前端在工程效能上已经做到非常的极致，极致到今天很多技术领域为了工程化的落地，就一定会去考虑和前端大的技术生态的连接。

另外，这句话描述的是杭州的自然景观，我也希望疫情能够早点散去，未来能够邀请大家来杭州做一些面对面的交流。

好的，以上是本次 D2 大会主题的六大方向以及我本人的一些看法。最后，希望本次 D2 大会能够给大家带来一些收获，引发大家的思考。
谢谢大家！。

目录

语言框架 **6**
ReScript: JavaScript 平台上高质量大规模程序构建的秘密武器 6
2022 年 Deno 的生态状况介绍 16

跨端技术 **28**
使用跨端方案 Rax 编写鸿蒙应用 28
V8 JS AOT 化的探索与实践 42
浅谈 web component 实践与应用 55

Serverless **67**
云原生 WebAssembly 与 Serverless 67
Node.js Web 框架再进化 — 面向前端与未来标准 81
用 Rust 和 Node-API 开发高性能 Node.js 模块 94

前端体验 **112**
基于大数据的前端页面异常结果检测 112
用户行为链路的识别、挖掘及诊断 123
适老化&无障碍的前端体验探索 — 为更多人提供便利和自由 141

低代码 **165**
腾讯低代码引擎建设思考和实战 165
ESM Bundleless 在低代码场景的实践 188
面向中后台复杂场景的低代码实践思路 203

多样化 **219**
解读千万级关系网络下的数据探寻可视化技术 219
向量到墨水：从绘画视角解析三维模型的漫画风格渲染技术 238
开源表单方案 Formily 的核心设计思路 252
协同文档工作机制简介 266
CRDT 实时协作技术在稿定编辑器中的应用 279
虚拟偶像诞生记 — 数字人行业和技术探究 287
Web 端短视频编辑器的设计与实现 — 像做 PPT 一样做视频 309

语言框架

ReScript: JavaScript 平台上高质量大规模程序构建的秘密武器

摘要：ReScript 是第一个完全由中国人主导、并在世界范围内有广泛用户的通用程序语言。它放弃了对 JavaScript 语法的完全兼容，保证了单个模块级别兼容。它专注于解决实际问题，优化工具链的性能和用户体验，其编译器和构建系统的性能超出了 TypeScript 一到两个数量级，其类型信息也用于辅助生成更高效的代码。

分享人：张宏波，ReScript 编程语言作者，前 OCaml 程序语言的唯一中国核心开发人员。Meta 唯一一位在华特聘的软件工程师。本科毕业于清华大学电子工程系，在 MSRA 实习期间对函数式程序语言产生了浓厚的兴趣，被沈向洋院士推荐赴宾夕法尼亚大学就读程序语言的博士学位。博士第二年受 Bloomberg 破格邀请以硕士毕业的身份负责领导该公司函数式语言编译器的开发和维护工作。在该公司工作期间开源的 BuckleScript 编译器是当时该公司最受欢迎的开源软件，多次被公司官方报道。

目录：

一、ReScript 简介和发展史
二、ReScript 基于数据的编程
三、示例：红黑树

一、ReScript 简介和发展史

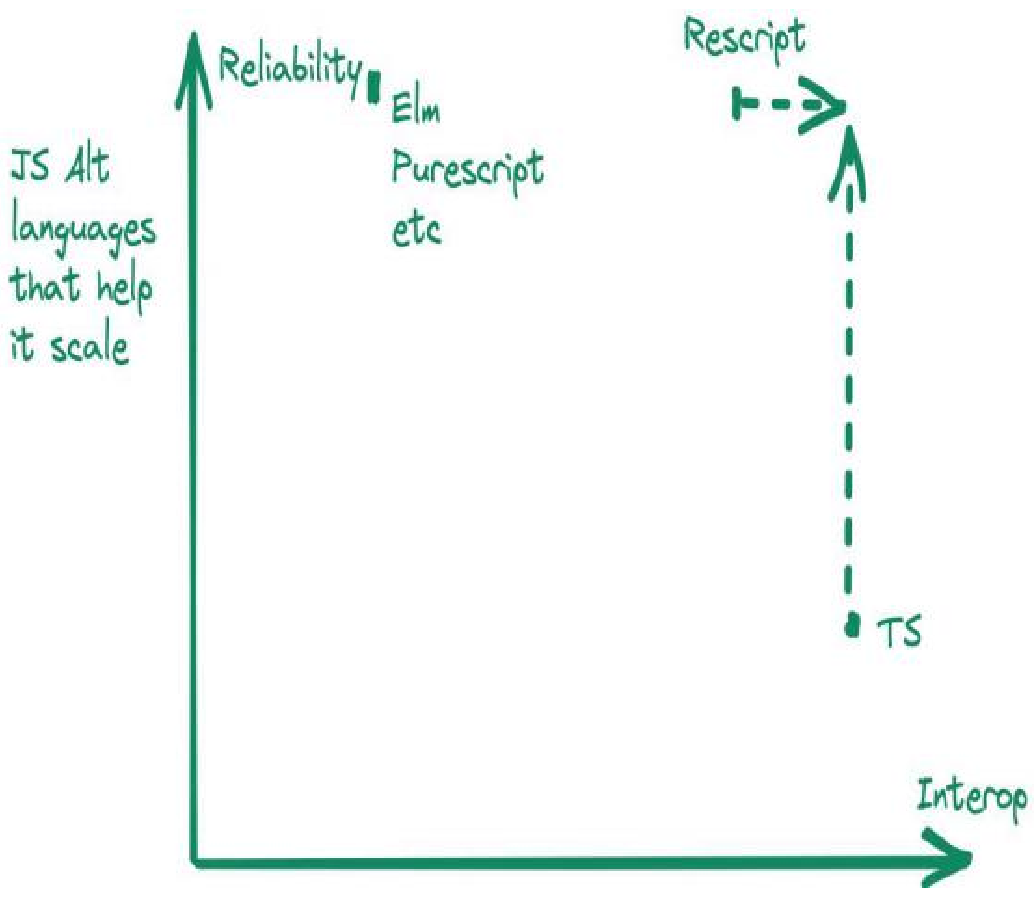
ReScript 是在放弃对 JavaScript 的语法完全兼容，同时保证单个模块级别的兼容的情况下一门 JavaScript 平台上的函数式语言，其类型系统和模块化编程的思想积累了学术界近三十年的研究经验。和传统的学术语言不同的是，ReScript 摒弃了一些学院派的思维，专注于解决实际问题，优化工具链的性能和用户体验，其编译器和构建系统的性能超出了 TypeScript 一到两个数

量级，其类型信息也用于辅助生成更高效的代码。ReScript 是第一个完全由中国人主导并在世界范围内用于商业开发的通用程序设计语言。

1、ReScript：一种静态类型的函数式语言

ReScript 用来解决 JavaScript 大规模代码可依赖的重构问题。JavaScript 是脚本语言，大规模代码的重构是很有挑战性的问题，而解决重构的经典方案就是引入一个静态的类型系统。

下图针对与 JavaScript 的互操作性（横轴）和类型性可依赖性（纵轴）把 ReScript 和其它编程语言进行了对比：



- Elm 或 PureScript 完全放弃了与 JavaScript 的互操作，设计一个封闭的语言；
- TypeScript（TS）是另外一个极端，兼容了所有的 JavaScript，然后慢慢变得更安全；
- ReScript 则属于折中的方式，即在不牺牲类型性可依赖性的前提下与 JavaScript 的互操作作越来越好；

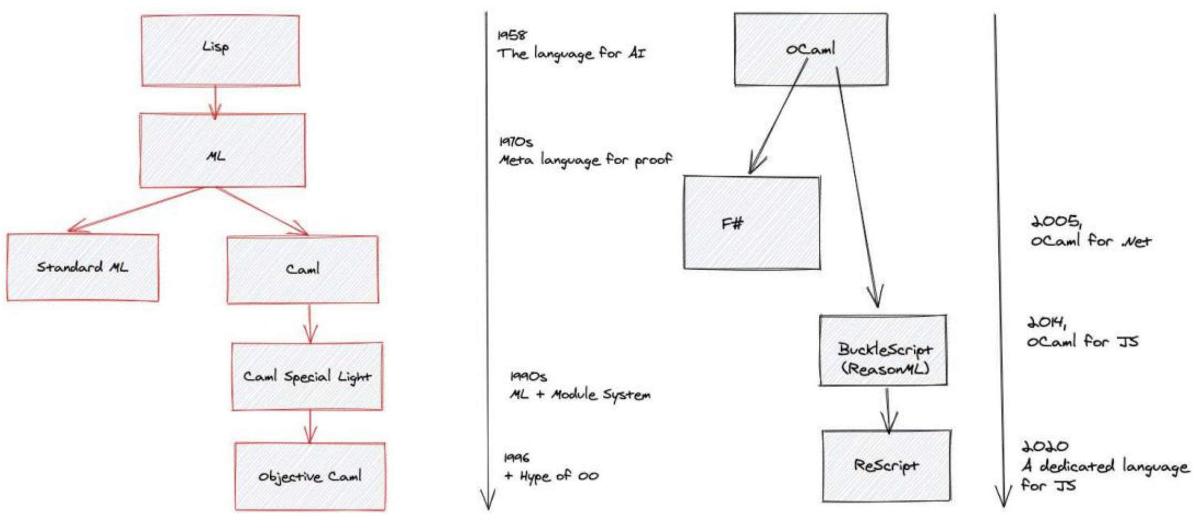
因此，TS 和 RS 是从两个角度使 JavaScript 代码重构越来越可靠，RS 是在不牺牲安全的情况下尽可能和 JavaScript 友好的互操作，TS 则是在完全互操作的情况下越来越安全。

2、ReScript 的特点：

- ReScript 是对 JavaScript 平台的优化；
- 起源于学术界，最早是一个 ML（meta-language），将学术界好的概念搬运到工业界；
- 重点关注工程的实现，让编辑器越来越快，生成的代码越来越高效。

3、ReScript 的发展历史

ReScript 是根源于学术界，下图展示了 ReScript 的学术脉络：
1958 年为了解决 AI 提出了最早的函数式语言 LISP，70 年代出现了 meta-language（ML）元语言，在法国 ML 分成 Standard ML 和 Caml，Caml 发展到 Caml Special Light，1996 年加入了 OO 代码变成 OCaml（Objective Caml），ReScript 则是从 OCaml 派生而来。



ReScript 的学术脉络

4、ReScript 的国际影响力

ReScript 是第一个真正意义上中国人创造的具有海外影响力的通用编程语言，用户遍布世界各地，拥有除英语以外的语言版本包括法语、德语、日语、韩语、西班牙语等。在韩国用的比较

多，在巴西甚至还开设了培训班。

韩国 Rescript 网站：https://green-labs.github.io/

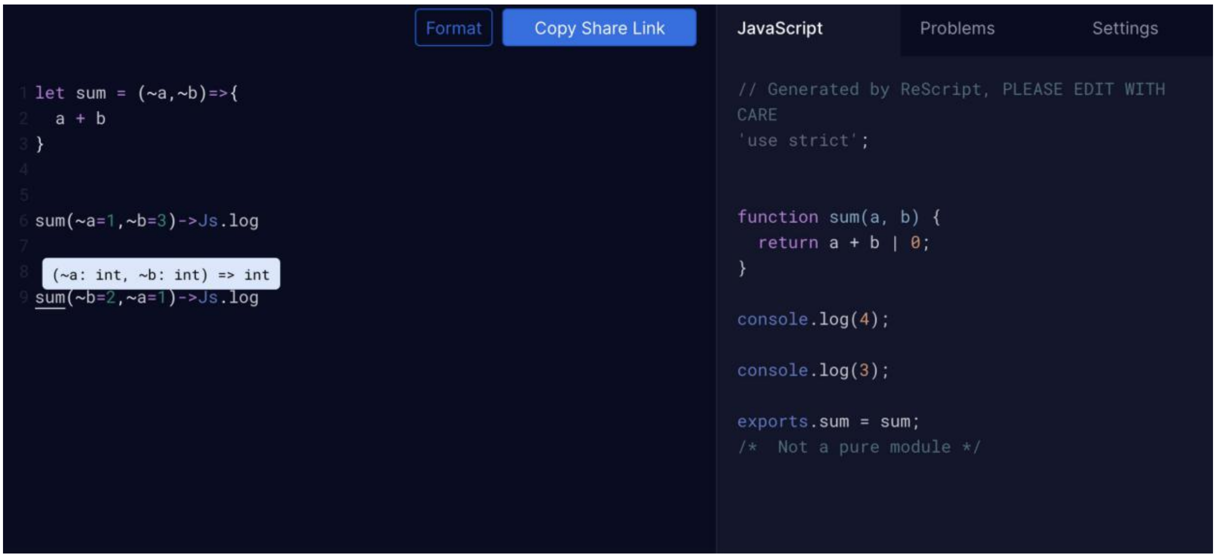
有人曾在知乎上提问：为什么中国出了那么多厉害的互联网公司，但没有自己设计过编程语言？我认为创造一个编程语言并不难，但让大众信任和使用这个编程语言是一项很困难的任务。ReScript 最初的 demo 也只用了六个周末，但将它做成一个具有生产价值的编辑器却用了六年时间，而且到现在还有很多需要改进的地方。



5、ReScript 的特性

下图是一个简单的 ReScript 示例，左边是 ReScript 代码，右边是生成的 JavaScript，通过示例可以看到 ReScript 的特性包括：

- 支持给函数名加 label，提高可读性；
- 自动进行函数类型推断，用户无需标注类型；
- 编辑器自动优化；

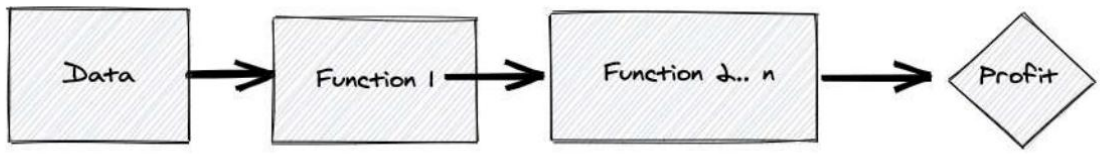


二、ReScript 基于数据的编程

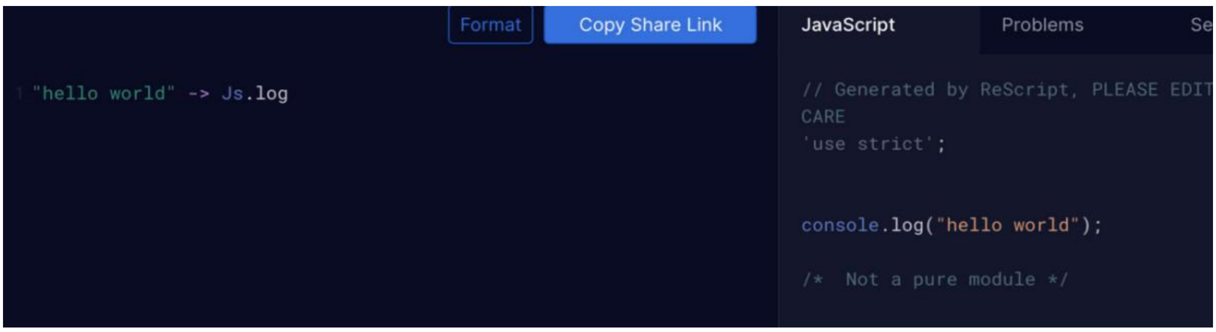
数据编程的核心：

1、数据和方法分离：

即单独定义数据，然后编写函数来操作数据，最后生成新数据。



如下图示例，左边“hello world”是数据，Js.log 是函数，→箭头是 pipe，右边是生成代码。

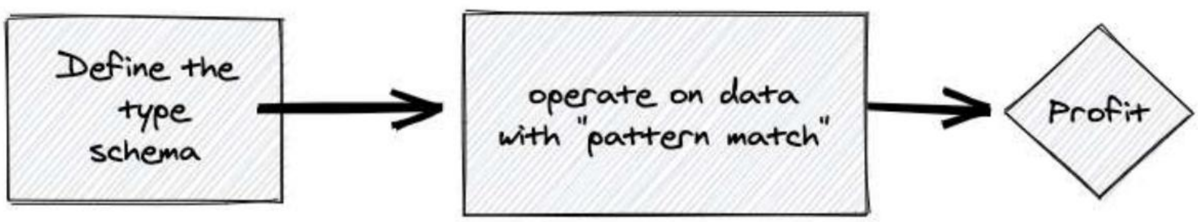


下图是一个相对复杂一点的示例，数据是 1,2,3，有三个函数 map、keep、reduce，open 相当于 JavaScript 的 import。

```
open Belt.Array
[1,2,3]
-> map(x=>x+1)
-> keep(x=> x > 2)
-> reduce((a,b)=>a+b)
-> Js.log
```

2、模式匹配

首先定义一个 type schema，然后通过模式匹配解析数据，最后生成新数据。type schema 相当于一个数据描述语言，其表达能力决定了数据处理能力；pattern match 相当于对数据的解析，通过解析将要素重新绑定生成新数据。



下图是一个 type schema 示例，白色框中是对应的 TypeScript 的定义，左侧是 ReScript。对比 ReScript 和 TypeScript 的写法有点类似，但二者最大的区别在于 TS 不可将任意类型进行 union，这个示例中如果类型有交集就不能用 number 和 undefined 这样定义，而 ReScript 则支持任何类型的 union。

```
type maybeInt =
    | Some (int)
    | None

let f = (x)=>{
  switch x {
    | Some (x) => x
    | None => 0
  }
}
```

```
type maybeInt =
  | number
  | undefined

var f = (x : maybeInt)=>{
    if(typeof x === "undefined"){
        return x
    }
    return 0
}
```

```
// Generated by ReScript, PLEASE EDIT WITH
CARE
'use strict';


function f(x) {
  if (x !== undefined) {
    return x;
  } else {
    return 0;
  }
}

exports.f = f;
/* No side effect */
```

下图是一个单项列表示例，可以对比下 ReScript 和 TypeScript 的写法，对于一个递归的数据类型的写法，同样这里 TypeScript 如果 number 和 undefined 有交集的话也不能这么写。

```
type rec list =
    | Cons (int, list)
    | Nil
let rec map = (x,f)=>{
  switch x {
    | Nil => Nil
    | Cons(x,xs)=> Cons(f(x),map(xs,f))
  }
}
```

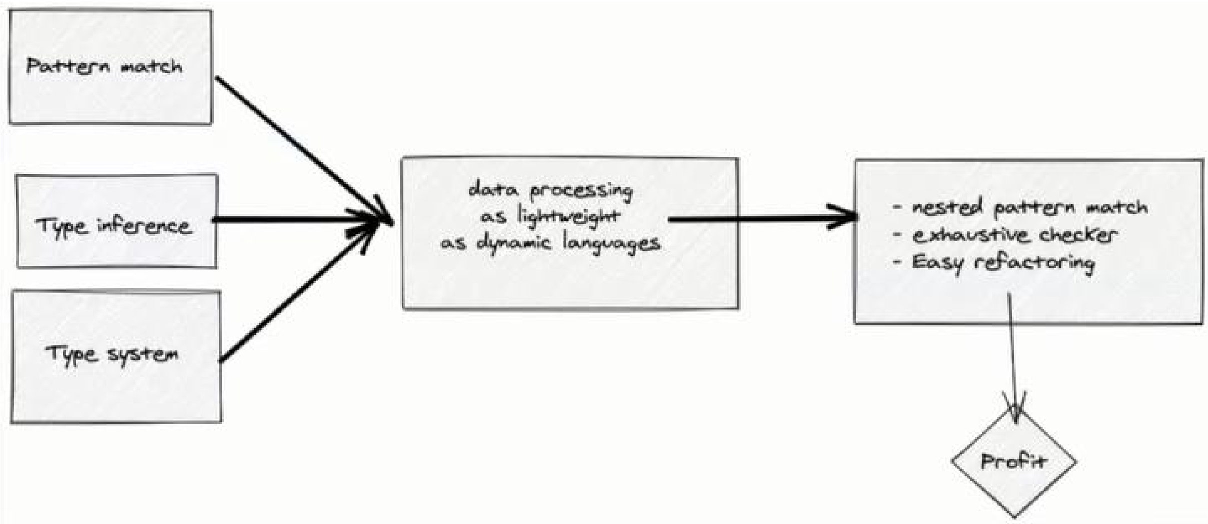
```
type list =
    | [ number , list]
    | undefined

var map = (x : list, f : (n:number)=> number) : list => {
    if (typeof x === "undefined"){
        return undefined
    }
    var [x0,xs] = x
    return [f(x0),map(xs,f)]
}
```

3、ReScript 是模式匹配语言

基于数据的编程的核心要素在于模式匹配，它可以：

- 对任意类型进行组合，支持深度嵌入；
- 对模式进行详尽检查，如漏网模式检测；
- 进行类型推断，无需用户标注类型；



模式匹配

三、示例：红黑树

1、如何定义：type

红黑树是一个平衡二叉查找树，分左右两边，其节点有红黑两种颜色，首先要定义颜色 R 和 B，节点中有叶子节点和中间节点两种分别定义；

```
type color =
  | R
  | B

type rec tree =
  | Leaf
  | Node({val: int, l: tree, r: tree, color: color})
```

下图是对比 TypeScript 的定义

```
type color =
    | 'R'
    | 'B'

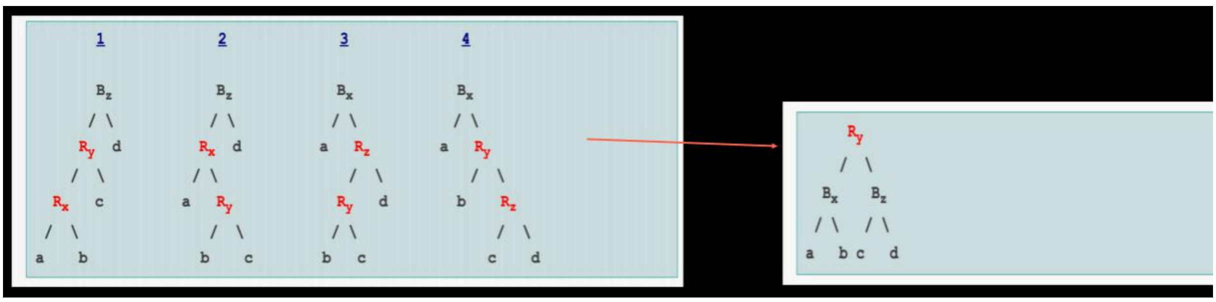
type leaf = undefined
type tree =
    | leaf
    | { val : number, l : tree , r : tree , color : color}
```

2、认识红黑树

红黑树有红色和黑色节点，根节点是黑色，红黑树的高度是黑高度即从根节点到叶子节点的黑色节点个数。之所以称为平衡二叉树是所有红黑树的黑高度相同。另外，为了保证红黑树的平衡性，相邻的两个节点不能同时为红色。

3、红黑树在 ReScript 中的实现：旋转

如果插入的节点为红色，则有两种情况，一种是相邻的节点为黑色，符合红黑树的规则可以停止；另一种是相邻的节点为红色会有四种可能的不规则模式（如下左图，分别为：左左、左右、右左、右右），为了避免红节点相邻需要将其旋转，运用函数式数据的旋转方法，将四种可能性最终旋转成右图的结构，现在根节点为红色，当红色节点到达最上面就会变成黑色完成旋转。



在 Rescript 中的具体实现如下图，4 个 Node 是上面左图中红色的四种可能的模式，val:z 对应了 Bz 就是模式 1，这里精确的描述了模式 1 的节点，另外 3 个 Node 也同样。这里得到一个巨大的模式匹配，有或模式，有嵌套模式，且可以无限嵌套，这是 ReScript 与 TypeScript 的不同，ReScript 是基于数据的编程模式，可以提供模式解构数据并得到新的数据。

```
let rotate = tree => {
  switch tree {
  | Node({
    val: z,
    l: Node({color: R, val: y, l: Node({color: R, val: x, l: a, r: b}), r: c}),
    r: d,
  })
  | Node({
    val: z,
    l: Node({color: R, val: x, l: a, r: Node({color: R, val: y, l: b, r: c})}),
    r: d,
  })
  | Node({
    val: x,
    l: a,
    r: Node({color: R, val: z, l: Node({color: R, val: y, l: b, r: c}), r: d}),
  })
  | Node({
    val: x,
    l: a,
    r: Node({color: R, val: y, l: b, r: Node({color: R, val: z, l: c, r: d})}),
  }) =>
    Node({
      color: R,
      val: y,
      l: Node({color: B, val: x, l: a, r: b}),
      r: Node({color: B, val: z, l: c, r: d}),
    })
  | _ => tree
  }
}
```

另外，AVL 树也可以用 ReScript 通过模式匹配实现，有兴趣的朋友可以参考知乎文章《20 行 ReScript 代码实现 AVL 树》。

《20 行 Rescript 代码实现 AVL 树》网址：https://zhuanlan.zhihu.com/p/430867594

ReScript 是一个很有深度的语言，本次仅分享了其中一部分，感兴趣的朋友欢迎到官网了解更多内容。

ReScript 官网：https://rescript-lang.org/

2022 年 Deno 的生态状况介绍

摘要： 本话题 Bartek 将为你介绍 2021 年 Deno 的新特性和路线图，2022 年整个 Deno 的生态状况，以及在现实世界中选择 Deno 的好处。

分享人： Bartek Iwańczuk，Deno 核心开发者，全栈工程师，2019 年加入 Deno 团队，主要负责 Deno CLI 的相关工作。

一、什么是 Deno?

Deno 是一个崭新的 command-line runtime，可执行 JavaScript 和 TypeScript，它类似于 Node.js，但 Deno 允许执行 TypeScript 文档。

Deno 的目标是有趣和高效，我们相信一个好的脚本平台对于接受当前状态的工具来说太重要了。

2018 年 Deno 在 Ryan Dahl《Node.js 十个让我后悔的事情》的知名演讲中首次被介绍，2 年之后，Deno V1.0 诞生，目前是 1.17 版本，开发基础已相当高，我们在持续不断地发布新版本。

Deno 的亮点：

1、Deno 是一个单一的可执行文件，而且他始终如此；
2、只需要下载一个文件，就可以运行任何 Deno 程序；
3、Deno 适用于 Linux、Mac 和 Windows；
4、支持开箱即用的 TypeScrpit；
5、TypeScript 编译器实际上是 Deno 程序的一部分，让 Deno 可以执行您下载的脚本，我们使用 V8 Snapshot 的特性来快速启动编译器；
6、TypeScript 编译器还使用一个名为 SWC 的工具，它是一个基于 Rust 的 JavaScript 和 TypeScript 解析器。实际上它不仅是一个解析器，而且还提供了更多的功能；

7、Deno 默认有一个称为安全执行的东西，这意味着您的程序不能访问文件系统、网络连接，除非你明确允许，否则它们无法暂停子进程或访问环境变量。
例如我们从远程 URL 运行一个程序：

```
> deno run https://deno.land/std/examples/welcome.ts
Welcome to Deno
```

这是 Deno 的功能之一，也很有趣，你无需事先下载资源，这一切都有运行时为你完成。

在这里我们正在下载一个名为 Welcome 的程序，这也是我们标准库的一部分，Deno 会下载资源，并从 TypeScript 转换成 JavaScript，然后执行它。

如果要允许 Deno 可以访问文件系统、互联网连接、子进程或环境变量，你需要通过提供所谓的允许标准来让 Deno 可以这样做：

```
—allow-read=/tmp
—allow-net=google.com
```

—allow-read=/tmp，意味着你的程序现在可以读取您的文件系统中的/tmp 目录；
—allow-net=google.com，是指 Deno 能够访问互联网连接，但只能连接到 Google.com 域。

8、Deno 可以从 URL 访问资源，无需事先在本地下载它们，在这里你可以看到，我们使用了一个名为 magenta 的函数（它也来自 Deno 标准库）：

```
import { magenta } from “https://deno.land/std/fmt/colors.ts";
```

它来自一个名为 fmt color 的模块，URL 解析实际上是与浏览器相同的，因此，这与使用 ES 模块时，浏览器将使用的 URL 解析算法完全相同。

9、Deno 没有像 Node 那样的 node_modules 或 index.js 概念，所以它不会让您的文件系统寻找匹配的包，这并不完全正确，我们稍后讨论。

10、由于使用了 URL，Deno 不像 NPM 那样依赖于一个集中式服务器，你实际上可以从 internet 上的任何公共服务器下载源代码，所以你可以使用像 Skypack、UNPKG、ESM 这样的 CDN，或者使用任何您自己的服务器，如果您想在那里托管您的资源。

11、再强调一次，运行程序所需的一切都有 Deno 处理，你不需要单独的包管理器，Deno 遇到没有访问过的 URL 时，它会请求获取资源，然后将它们缓存在一个全局缓存目录中，并将资源转换为 JavaScript。因此，下次您尝试访问已经访问过的 URL 时，Deno 不会发生请求获取资源，而会使用本地缓存。

12、Deno 大量使用 Web 标准 API，在服务器环境中，使用 Web API 来包装是合理的，Deno 更愿意使用 Web 标准 API，而不是一个 Deno 专有的新 API。

例如有一个名为 fetch 的常用 API，您可以使用它来下载或发送实际请求到任何 URL，因此，与其想出一个新的 API 来做到这一点，不如选择复用在大部分浏览器都已经可用的 fetch API。

Deno 还支持许多 API，如下表，但并非所有，例如新的 Web Crypto API 已在 Deno 中，但也不是全部都支持，我们正在努力中。

- fetch
- Web Crypto
- Web Streams
- Web Workers
- Web Storage
- Web Sockets
- AbortController
- WebGPU
- Blob
- Location
- EventTarget
- TextEncoder
- TextDecoder

- onload,
- prompt(), confirm(), alert()
- many more...

Deno 也支持标准的 Web Workers API，支持 AbortController，甚至提供了完整的 WebGPU 支持，因此您实际上可以直接在 Deno 中运行您的机器学习模型，并在显卡上非常高效的运行。

为了确保所有的这些 API 的实际工作方式与它们在浏览器中的工作方式相同，Deno 使用了一种叫做 Web Platform Tests 的东西，对所有浏览器进行大量测试，以确保它们正确的符合标准。

因此，Deno 现在被列在 MDN 的兼容性表中，任何时候，您去 MDN 寻找一些 JavaScript API 的适用性时，如果 Deno 也支持它，您会在右侧表中看到它的适用信息。

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	WebView Android	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	Deno	Node.js
`setTimeout`	30	12	1	4	4	1	4.4	30	4	10.1	1	3.0	1.0	Partial ★
Supports parameters for callback	Yes	12	Yes	10	Yes	?	Yes	Yes	?	?	?	Yes	1.0	Yes

Deno 现在被列在 MDN 的兼容性表中

13、Deno 不仅仅是一个 runtime，它实际上是一个完整的工具链，在一个二进制文件中提供了整个工具链，当您开始一个新项目时，不需要去寻找测试器或格式化工具，也不需要单独安装它们，创建项目时，可能所需的一切都已经存在在 Deno 里了。

```
SUBCOMMANDS:
bundle Bundle module and dependencies into single file
cache Cache the dependencies
```

```
compile UNSTABLE: Compile the script into a self contained executable
completions Generate shell completions
coverage Print coverage reports
doc Show documentation for a module
eval Eval script
fmt Format source files
help Prints this message or the help of the given subcommand(s)
info Show info about cache or info related to source file
install Install script as an executable
lint UNSTABLE: Lint source files
lsp Start the language server
repl Read Eval Print Loop
run Run a JavaScript or TypeScript program
test Run tests
types Print runtime TypeScript declarations
upgrade Upgrade deno executable to given version
```

例如 Deno 附带了一个 bundler，可利用 Deno bundle 子命令来使用；还有一个 Deno fmt 的子命令用来处理代码格式化，使用方式与 Prettier 非常相似，但它的速度更快；如果你想在系统上安装一些源代码作为可执行文件，您可以安装 Deno，或者可以使用 Deno LSP 与 IDE 和代码编辑器进行集成。Deno 内置了一个非常简单的测试运行器，它能大致符合您的预期需求。

编写测试从未如此简单，您可以在与代码相同的文件中编写测试，然后执行它们也不会出现问题。

Demo 演示：

我们编写一个只有五行的简单程序，该程序将在访问我们指定的 URL 后打印服务器标头。

```
import { magenta } from "https://deno.land/std/fmt/colors.ts";
const url = Deno.args[0];
```

```
let res = await fetch(url);
let server = res.headers.get("server")!;
console.log("server:", magenta(server));
```

- 在打印之前，我们会为标题名称着色，首先我们从导入 magenta 函数开始，同样，这次我们使用 Deno 标准库的 fmt colors 模块，这是一个标准的 ES 模块语句，其工作方式与在浏览器中的工作方式相同；
- 然后我们获取传递给程序的第一个命令行参数，在这里您可以看到我们用到了这个 Deno 命名空间，因此，访问命令行参数不是您可以在浏览器中所做的事情，有没有可以复用的 Web API，这就是为什么会需要 Deno 命名空间。
- 在获取 URL 后，我们将需要获取资源，在这里我们使用了 fetch API，同样，fetch 在浏览器中可用，所以 Deno 复用了这个 API，而不是引入一个新的 API；在这里你还可以看到我们使用了 await 关键字（置于 fetch 前），所以 Deno 支持前置的 await；
- 在获取 URL 的资源后，我们需要取得响应的标头，响应的标头，这又是一个 Web 兼容的 API，就像你在浏览器中所做的，在 Deno 中所做的就应该一样；
- 最后我们将服务器标头记录到控制台，但首先，这个 magenta 函数会先执行。

下面是一些你在运行这个程序的示例参考：

```
$ deno run --allow-net curl.ts https://postman.com
Server: nginx
$ deno run --allow-net curl.ts https://www.cloudflare.com
Server: cloudflare
$ deno run --allow-net curl.ts https://github.com
Server: GitHub.com
```

可以注意到，--allow-net 这个没有值的标志，告诉 Deno 它可以访问互联网上的任何东西，当然我们可以在每个示例中将其限制，例如--allow-net =postman.com，然后 Deno 将只能访问 postman.com 域的 URL。

站在巨人的肩膀上

Deno 是一个全新的项目，并没有重造所有的轮子，而是建立在巨人的肩膀上，复用许多来自 Rust 生态系统的当红技术如下，但不仅仅是来自 Rust 生态系统。

```
- V8 (Chrome’s JavaScript VM)
- Rust
- TypeScript (Microsoft)
- Tokio (async I/O framework for Rust)
- SWC (JS/TS parser)
- Rustls (TLS implementation)
- Hyper and Reqwest (HTTP server and client libraries)
- wpgu (WebGPU backend)
- tungstenite (Websockets)
- trust-dns (Async DNS queries)
- rustyline (readline/REPL)
- notify (file system notifications)
- lspower (LSP framework for talking to IDEs)
- dprint (code formatter)
```

例如 JavaScript 执行引擎是 V8，和基于 Node 或 Chrome 或 Chrome-based 的浏览器一样；

Deno 中所有的编码都是使用 Rust 构建的，这与主要用 C++构建的 Node 不同，由于 Deno 支持 TypeScript，当然也使用来自 Microsoft 的 TypeScript；

Deno 建立在 Tokio 等技术上，类似于 Node 中的 libuv，但这实际上是一个用于执行异步 I/O 的 Rust 框架；

之前已经提到 SWC，它是 Deno 的重要组成部分，我们非常依赖其超快的 JavaScript 和 TypeScript 解析器，但正如我之前提到的，他实际上提供了更多的功能，比如 bundling，所以 Deno bundle 子命令实际上是在 SWC 之上实现的。

我们使用的是 TLS 实现的 Rustls，类似于 OpenSSL；

使用 wpgu，它是 WebGPU API 的后端，你可以在显卡上执行的所有操作实际上都是在此 API 之上实现的；

对于代码格式化 Deno fmt，我们使用 dprint，dprint 是一个与 Prettier 非常相似的项目，但同样，它不是在 JavaScript 中实现的，它是在 Rust 中实现的，所以它会更快。

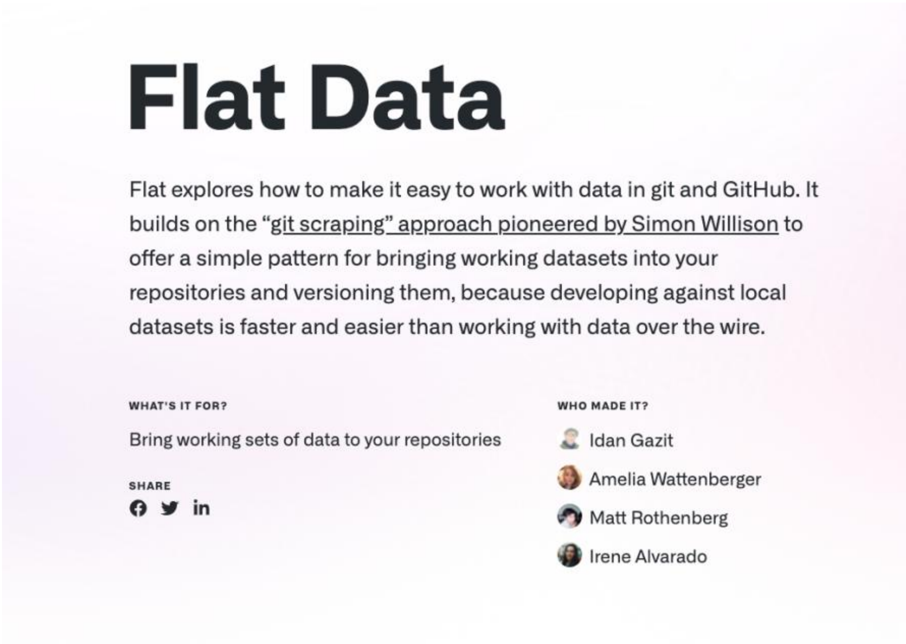
二、Deno 2022 年的生态状况

Deno 的发布非常频繁，我们用称为“火车模型”的方式来发布，每周都有发布，如果您错过了合并某些东西的时间，那么不需要担心，因为下周就会有另一个版本发布。

我们通常每个月发布 2-4 个补丁版本和一个次要版本，从 2020 年 5 月的 1.0 开始，每个月实施一个次要版本，所以到目前为止有 17 个次要版本。我们发布了大量功能和这些次要版本，以及近 100 个补丁版本，2022 年的计划是继续专注于 Web 兼容性，希望添加更多 Web API 工作，以运行更多功能和已支持的 Web API，例如 Web Crypto；我们也希望与 TypeScript 有更好的集成，虽然 Deno 已经能使用 TypeScript，但我们想进一步改进它，您可以在 IDE 中获得一流的 IntelliSense 和建议。

我们还想迭代我们的安全模型，可能会为其提供更多功能。正如我们在一些示例中看到的那样，Deno 有一个叫做标准库的东西，它是我们团队官方支持的一组模块，我们自己会进行审核和审查，并且保证这些 API 可以和 Deno 一起使用。标准库也会源源不断更新和增加新功能，因此希望我们能够在 2022 年稳定 Deno 的发布版本 1.0 以及所有标准库。

采用 Deno 的项目每月数据都在增加，例如有一个名为 Flat Data 的项目，那是一个 GitHub 项目，它使用 Deno fetch 来载入 URL 资源，基本上，它允许您在 GitHub 中运行 CI 期间执行单个文件（JavaScript）。因此在 GiHub 操作中，您可以拥有操作数据的工作流，这些数据提供给您的 GitHub 操作 API。



Flat Data

Flat explores how to make it easy to work with data in git and GitHub. It builds on the "git scraping" approach pioneered by Simon Willison to offer a simple pattern for bringing working datasets into your repositories and versioning them, because developing against local datasets is faster and easier than working with data over the wire.

WHAT'S IT FOR?

Bring working sets of data to your repositories

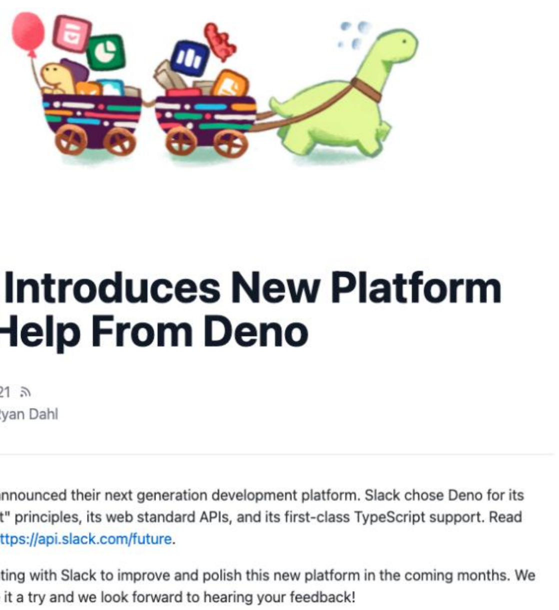
SHARE

WHO MADE IT?

Idan Gazit
Amelia Wattenberger
Matt Rothenberg
Irene Alvarado

https://next.github.com/projects/flat-data

就在最近，Slack 宣布其新的 API 平台将建立在 Deno 之上，然而它不会建立在 Deno CLI 之上，而是重用一些内部的 Deno 模块，建立一个符合 Slack 含义的平台。我们正在与 Slack 密切合作，以提供尽可能最好的体验。



Slack Introduces New Platform With Help From Deno

November 16, 2021
Aaron O'Mullan, Ryan Dahl

Today Slack has announced their next generation development platform. Slack chose Deno for its "secure by default" principles, its web standard APIs, and its first-class TypeScript support. Read more about it at https://api.slack.com/future.

We'll be collaborating with Slack to improve and polish this new platform in the coming months. We hope you will give it a try and we look forward to hearing your feedback!

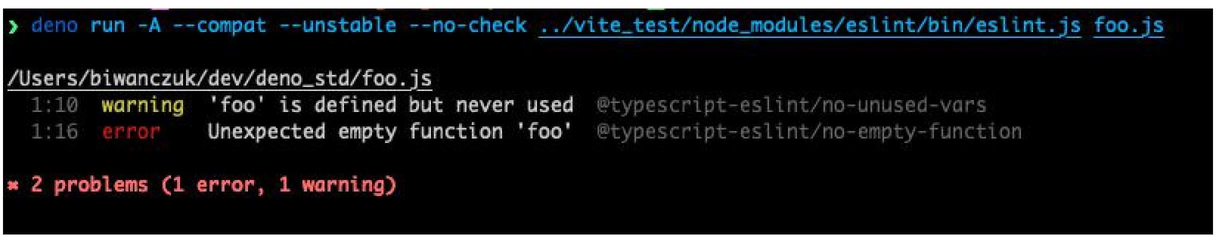
你还可以看到一些来自社区成员的 Twitter，比如 Jack Archibald，他们向您展示了 Deno 的能力以及如何与 Node 有何不同的 API 集，以及为什么可能更适合您的需求。



Deno 目前已经可适用于生产环境，但是我们听到一些意见，由于它与 Node.js 和 Node.js 生态系统有很大不同，因此现在很难被采用，我们也认知了这个问题，在整个 2021 年，我们花了很多努力，希望让 Deno 具备一流的 Node.js 兼容性，这意味着你能够将运行工具、程序和 Node.js 生态系统所编写的依赖项都运行于 Deno，我们称之为 compat 模式。

因此我们在过去两个月中一直在稳步前进，已经为众多内部节点模块提供了兼容性，目前仍在处理更大的模型，例如 HTTP、TTPS 或 zlib，我们预计将在 2022 年第二季度实现完全兼容性。我们这样做是为了更广泛的受众，尤其是能进入前端开发的生态系统，我们的目标之一是让热门的前端工具，如 Vite、Next.js 等，在和 Deno 工作中不会出现问题。

实际上，我们在这方面取得了很好的进展，并且已经有一些现有的 Node.js 工具可以在 Deno 中使用。



```
❯ deno run -A --compat --unstable --no-check ../vite_test/node_modules/eslint/bin/eslint.js foo.ts

/Users/biwanczuk/dev/deno_std/foo.ts
  1:10  warning  'foo' is defined but never used          @typescript-eslint/no-unused-vars
  1:17  warning  Unexpected any. Specify a different type @typescript-eslint/no-explicit-any
  1:21  error    Unexpected empty function 'foo'          @typescript-eslint/no-empty-function

✖ 3 problems (1 error, 2 warnings)
```

如上图，您可以直接在 Deno 中看到 ESLint 的示例。在这一点上它还不如 Node，因为 Deno 还没有完全支持 NPM，我们也在努力中，但这种兼容的方法很有价值，能让大多数用户使用。

有趣的是，即使我们提供了与 Node.js 的兼容性，所有的程序仍将遵循 Deno 的沙箱标准，除非你明确允许执行某些操作，否则 Node.js 程序将无法访问文件系统和环境变量或 Internet 连接。这实际上对现有生态系统来说是个巨大的优势，因为你将能够限制某些工具可以做什么或不能做什么。

三、关于 Deno 公司

Deno 开始时是一个开源项目，未来也将保持开源。Deno 是免费的，也是 MIT 授权的，您可以用于自己的使用目的。我们正在建立一家公司，为 CLI 以及相关项目提供更好的生态系统。我们已经有十名工程师，而且还在增长。现在我们正在开发的第一个商业产品是 Deno Deploy。

四、Deno Deploy

顾名思义，Deno Deploy 将允许您非常、非常轻松的部署您的项目，或者让您更好地了解它。Deno Deploy 是一个全球分布的 JavaScript 引擎，它在世界各地的众多数据中心运行，允许您在几分钟内将您的产品从一个想法交付到实际工作中。它还像 Deno 一样支持 TypeScript，因此您无需执行任何构建步骤即可交付您的产品。这是由运行时内部完成的，概念就像 Deno 的运行时一样，并且部署不是一个完全独立的项目，它实际上建立在与 Deno CLI 相同的工具之上，我们不必重造轮子，我们只是建立在已经运行良好的基础上，并进一步完善它。

那么部署可以做什么？这里有一个小例子。

BETA 3 / example-proxy-requests Open Editor

```
import { serve } from "https://deno.land/std@0.114.0/http/server.ts";

async function handler(req: Request): Promise<Response> {
  const url = new URL(req.url);
  url.protocol = "https:";
  url.hostname = "example.com";
  url.port = "443";
  return await fetch(url.href, {
    headers: req.headers,
    method: req.method,
    body: req.body,
  });
}

console.log("Listening on http://localhost:8000");
serve(handler);
```

https://deno.com/deploy

其中一项功能是为您提供 playgrounds，很像 CodePen，不同之处在于它们不在用户本地运行，这些 playgrounds 实际上部署在世界各地 Deno Deploy 支持的所有数据中心。你可以快速构建一个想法的原型，它将立即部署到世界各地，你甚至可以将这个 playgrounds 嵌入您自己的网页或通过链接分享给你的朋友。

当然，Deno 是开源的，我们接受贡献者和贡献代码，这些贡献非常受欢迎和赞赏，Deno 已经有超过 600 名贡献者，有很多很多项目可以参与，而 Rust 能力不是必须的。

在这里留下一个链接，它将向你展示 Deno 公司的 GitHub 上的所有项目，你可以找到自己感兴趣的东西并帮助提供新功能和改进现有功能。如果你想为开源项目做出贡献，请来找我们，我们很高兴为新贡献者提供所需的任何帮助。

网址链接：https://deno.land/manual/contributing

跨端技术

使用跨端方案 Rax 编写鸿蒙应用

摘要：华为的鸿蒙系统自发布以来，便广受关注。它支持 JavaScript 编写应用的特性，对于前端开发者来说意义非凡。跨端解决方案 Rax 的团队也于 8 月份开始与 HarmonyOS JS UI 框架团队合作，旨在支持使用 Rax 编写 HarmonyOS 应用。经过几个月的实践之后，他们决定将内部细节的技术实现带到 D2 的舞台上与大家分享，让我们人手都能开发一款属于自己的 HarmonyOS 应用。

分享人：大果(刘雄)，阿里巴巴大淘宝前端工程师。

目录：

一、关于跨端解决方案 Rax
二、当 Rax 遇到鸿蒙系统
三、原理：Rax 是如何对接鸿蒙的

一、关于跨端解决方案 Rax

1、为什么会有跨端

在 PC 场景中，业务面对的是浏览器，针对性能的操作很有限，同时容器也是固定的。但是，在无线场景中之所以会出现各种端容器有以下三个原因：

- 设备&网络差异大带来的性能问题：典型场景比如淘特这种面向下沉用户，会更关注性能问题；
- App 容器层给了可能性：在无线端可以在容器层做更多定制能力；

- 大厂商的超级 App 生态：尤其在微信结合小程序成功打造自身生态后，更多大流量 App 也都纷纷基于小程序建设，因此小程序就成为一个必须选择的端。

在以上三方面的背景下，就引发了无线跨端的复杂问题。

2、跨端背景下需要什么样的技术方案？

跨端背景下的技术方案有 2 个思路：

- 一码多端：开发一套代码可以运行在多个端；
- 各写一套：每个端开发一套代码，常见的手机应用就是使用这种方案；

下图中通过研发成本、适配成本、功能一致、招聘成本这四个维度对两种方案进行对比。

- 研发成本：一码多端只需要开发一套代码显然成本会低；
- 适配成本：由于不同端的适配非常复杂，一码多端的成本就会随之升高；
- 功能一致：即业务逻辑的一致性，一码多端由于只有一套代码易于保障功能一致性，各写一套则比较难保障，比如常见的一个功能在 Android 可以使用但 iOS 不能；
- 招聘成本：一码多端只专注一种代码，相对人员的学习成本也较低。

	一码多端	各写一套
研发成本	✓	✗
适配成本	✗	✓
功能一致	✓	✗
招聘成本	✓	✗

基于以上对比，大前端或大体量的业务模式基本会选择一码多端的方式。一些有特别需求的场景比如交易场景，由于对稳定性的要求比较高，就会选择各写一套的模式。

3、社区中一码多端的方案

Rax、Remax、uni-app、Taro 等。

4、Rax 是一个怎样的跨端方案？

Rax 使用 React 语法编写 Web 应用，并且提供上层配套工程框架体系。在跨端能力方面，Rax 支持 Web、各厂商小程序、Weex 等不同端。目前 Rax 服务手淘内 80% 以上的业务，Rax 核心服务于阿里整体无线业务体系。



二、当 Rax 遇到鸿蒙系统

1、关于鸿蒙系统

鸿蒙系统是华为自研开发的国产操作系统，整体装机量突破 1.5 亿。目前鸿蒙系统采用兼容安卓的模式，但在未来，鸿蒙有机会跟安卓、iOS 三足鼎立。

鸿蒙系统支持使用 JavaScript 开发系统应用，这是安卓或 iOS 都很难触达的。



鸿蒙系统的架构分层

2、鸿蒙系统与跨端的结合

如下表所示，鸿蒙系统上端提供两种应用开发的解决方案：Java 和 JavaScript。Java 开发在未来主要是支持 JavaScript 的开发。JavaScript 提供两种模式：类 Web 范式和声明式，前者更多会面向大前端生态，而后者更多会面向客户端的开发。

鸿蒙系统将 JavaScript 作为应用开发的一等公民，因此，Rax X 鸿蒙具备理论可能性。

比较项	Java UI框架	方舟开发框架	
语言生态	Java	JS	eTS
接口方式	命令式	类Web范式	声明式
执行方式	开发者处理，基于API驱动的UI变更	框架层处理，基于数据驱动的UI自动变更	框架层处理，基于数据驱动的UI自动变更
相对优势	UI元素更丰富，开发更灵活	轻量化，开发更简便	极简开发，内存占用更少、运行性能更高

3、业务的诉求

- 单独开发一个鸿蒙应用

由于目前鸿蒙系统提供安卓兼容模式，对于业务来说，单独开发一套针对鸿蒙的应用系统成本过高，不具备普适性；

- 开发一个有流量入口的服务卡片

服务卡片是鸿蒙系统主推的能力，鸿蒙会分发公域流量给业务使用，有效解决业务层面流量不足的问题，十分符合业务需求。

因此，开发一个有流量入口的服务卡片成为阿里以及很多公司的业务场景。



鸿蒙服务卡片示例

4、技术方案

业务在技术层面的问题，首先是上面提到的单独开发一套应用的成本太高，另外，新技术太多已经学不动了，而使用 Rax 去开发鸿蒙应用的优势包括：

- 使用类 React 语法开发，不需要学习新的语法；
- 一套代码多个端，降低开发成本；
- 开发链路上可以更好的结合 Web 生态；

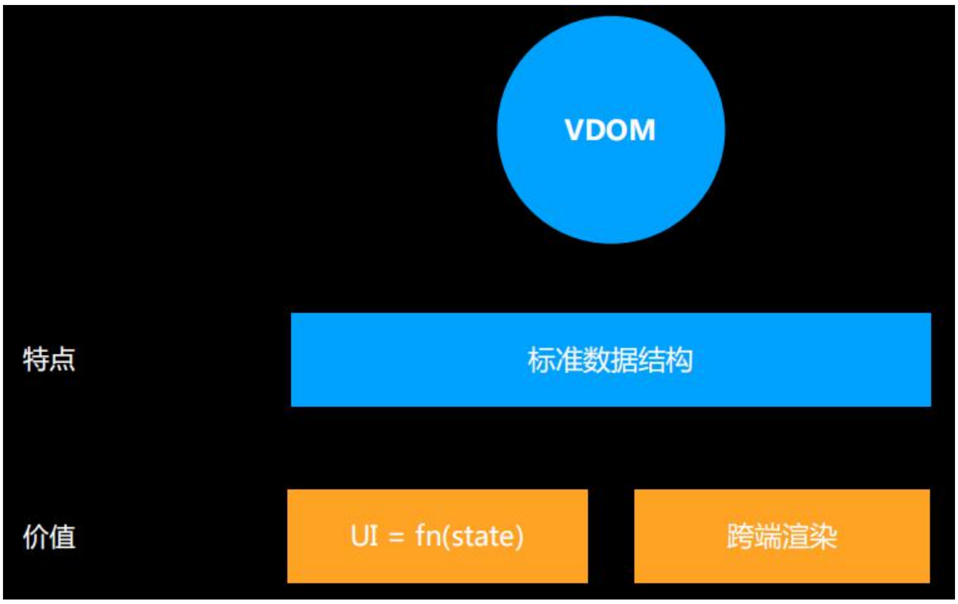
三、技术原理：Rax 如何对接鸿蒙应用

Rax 对接鸿蒙方案分三部分内容：渲染能力、工程构建和研发链路。

1、多端渲染

VDOM 的跨端能力

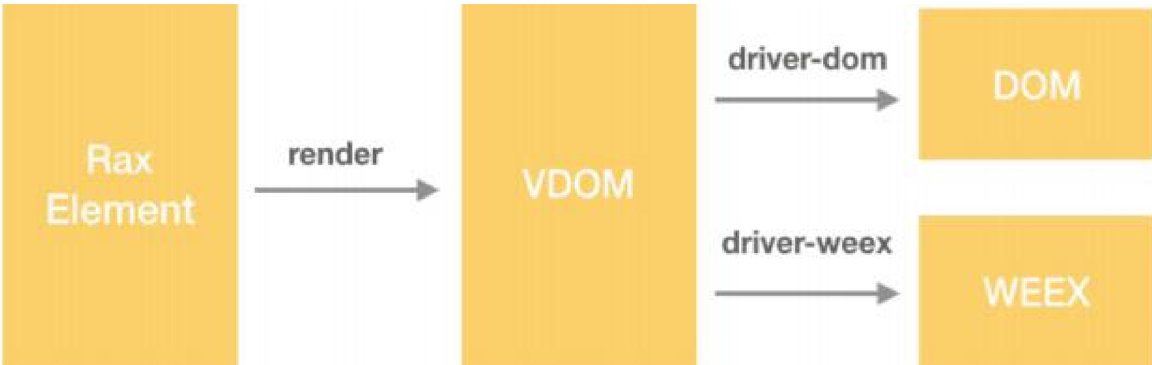
VDOM 的核心特点是标准的数据结构，可以跟具体的端解耦，进而更好的支持跨端渲染；另外，抽象一层标准数据结构能更有机会做到数据驱动 UI 的编程范式。



Rax Driver 适配器

在 Rax 中利用 VDOM 进行多端渲染的核心是 Rax Driver 适配器模式。

在与端耦合的逻辑中，都会抽象一个 Driver，因此会有面向浏览器的 driver-dom、面向 WEEX 的 driver-weex，以及小程序中的 driver 等。



下图中是 driver interface 的定义，声明与端绑定的方法，然后结合端的能力实现 api。

```
const Driver = {
  getElementById(id) {…
  },

  createBody() {…
  },

  createElement(type, props) {…
  },

  addEventListener(node, eventName, eventHandler) {…
  },

  removeEventListener(node, eventName, eventHandler) {…
  },

  removeAttribute(node, propKey, propValue) {…
  },

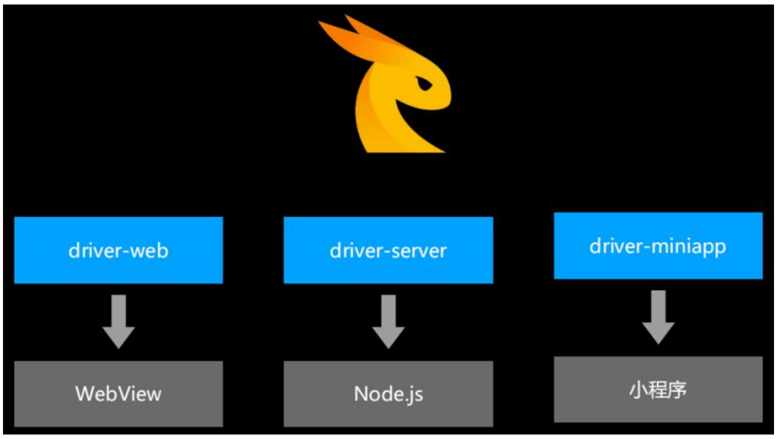
  setAttribute(node, propKey, propValue) {…
  },

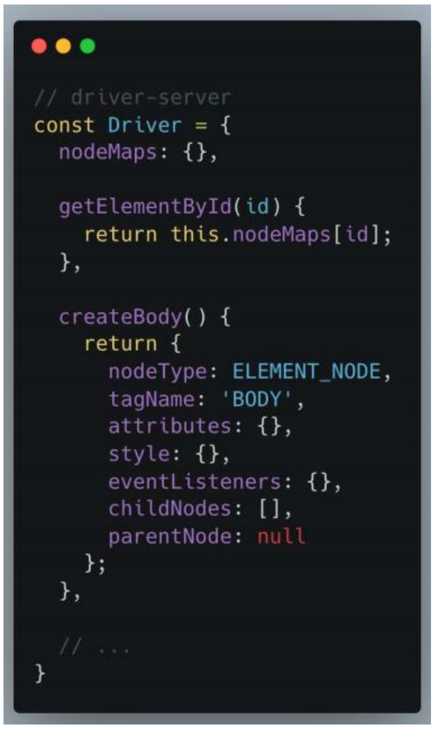
  setStyle(node, style) {…
  }
};

export default Driver;
```

Rax Driver 生态

在 Rax Driver 生态中有面向 WebView 的 driver-web，面向 Node.js 的 driver-server，面向小程序的 driver-miniapp。右图是一个简单的 driver-server 的 demo，可以看到在 server 端明确的 driver 和端解耦。





对接方案

鸿蒙官方提供的类 Web 开发范式是拆分了三个文件（见下图）：index.hml，index.css 和 index.js，写法与小程序/Vue 非常类似。



在这个背景下形成 Rax 对接鸿蒙系统的三个基本思路：

- 语法编译

逻辑：输入 Rax 语法，借助 Babel 工具进行语法编译，最终生成小程序 语法；

缺点：实现成本高，语法约束多；

- 类小程序运行时

逻辑：用上层 Rax 框架对接 set data 进行模板渲染；

优点：实现周期短；

缺点：运行时复杂度高，性能劣化；

- 对接鸿蒙底层渲染指令

逻辑：Rax 直接对接下层鸿蒙渲染指令创建节点；

优点：运行时轻量、高性能；

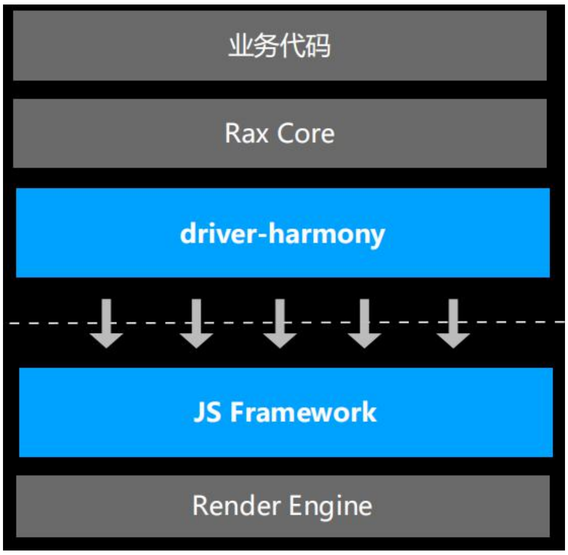
缺点：实现周期较长，需要对鸿蒙底层进行改造；



driver-harmony

下面左图为技术方案设计，最下面两层渲染引擎和 JS Framework 是鸿蒙提供的核心能力；上

面三层是代码层，包含：driver-harmony、Rax Core 和业务代码。其中核心部分是 driver-harmony 对接 JS Framework 的渲染指令，鸿蒙在 JS Framework 中提供跟浏览器标准类似的 API，在 driver 中调用 API 完成节点渲染。右图是一个简单的 demo，由于 API 与浏览器 API 存在差异，需要一些磨平操作。




```
const Driver = {
  // Internal state
  nodeMaps: {},

  getElementById(id) {
    return this.nodeMaps[id];
  },

  createBody() {
    return document.body;
  },

  createElement(type, props) {
    const node = document.createElement(type, {
      style: props.style,
    });

    this.setNativeProps(node, props, true);
    return node;
  },

  addEventListener(node, eventName, eventHandler) {
    node.addEvent(eventName, eventHandler);
  },

  removeEventListener(node, eventName, eventHandler) {
    node.removeEvent(eventName, eventHandler);
  },

  removeAttribute(node, propKey, propValue) {
    if (propKey === ID) {
      this.nodeMaps[propValue] = null;
    }
    return node.setAttr(propKey, undefined, false);
  },
};

export default Driver;
```

在实现 driver 后就可以实现 demo 链路的跑通，如下图，左边是鸿蒙页面的一个组件，在这个组件中需要在生命周期 onReady 里面调用 rax render 再传入对应的 driver-harmony，就可以实现在真机调试上渲染出 Rax 节点。

```
// pages/home/index.js
import { createElement, render } from 'rax';
import DriverHarmony from 'driver-harmony';

export default {
  onReady() {
    render(<text>Hello Rax</text>, {
      driver: DriverHarmony
    });
  },
  onDestroy() {
    console.info('Page onDestroy');
  }
}
```



2、工程构建

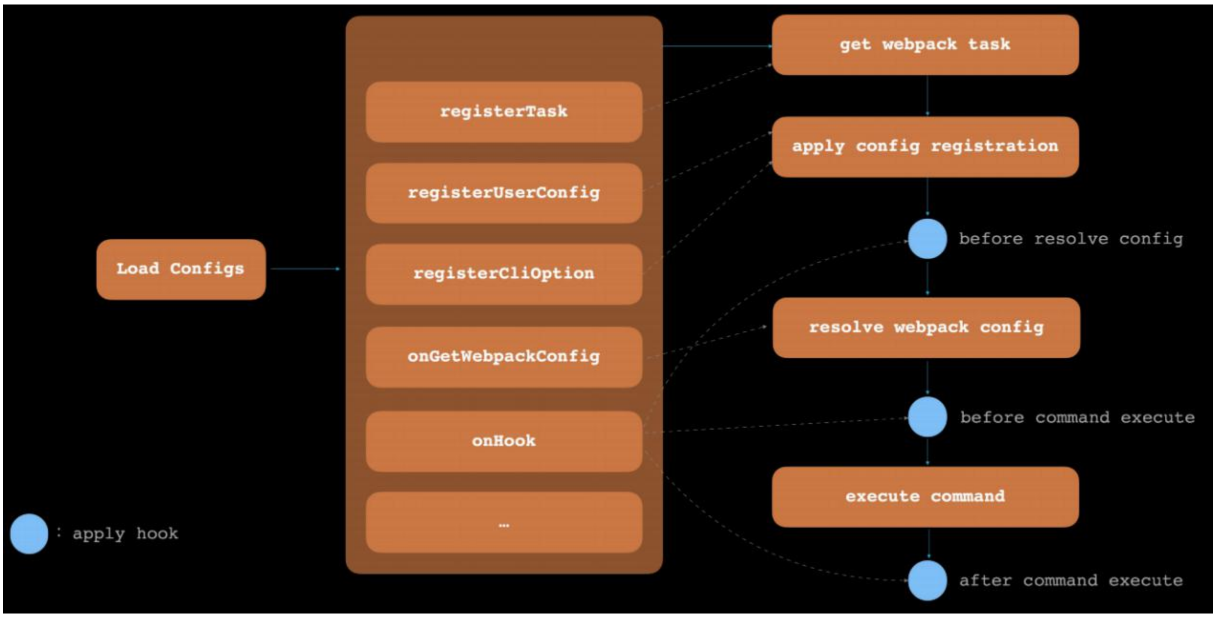
工程配置

在 Rax 面向多端的设计中有一个 targets，支持 web、miniapp 等不同的端，当鸿蒙进来后 targets 只需要加一个 harmony，然后再加入一个工程插件来处理鸿蒙应用构建的体系和 webpack 配置的体系。右图是配置多页面静态结构。



多 webpack 任务

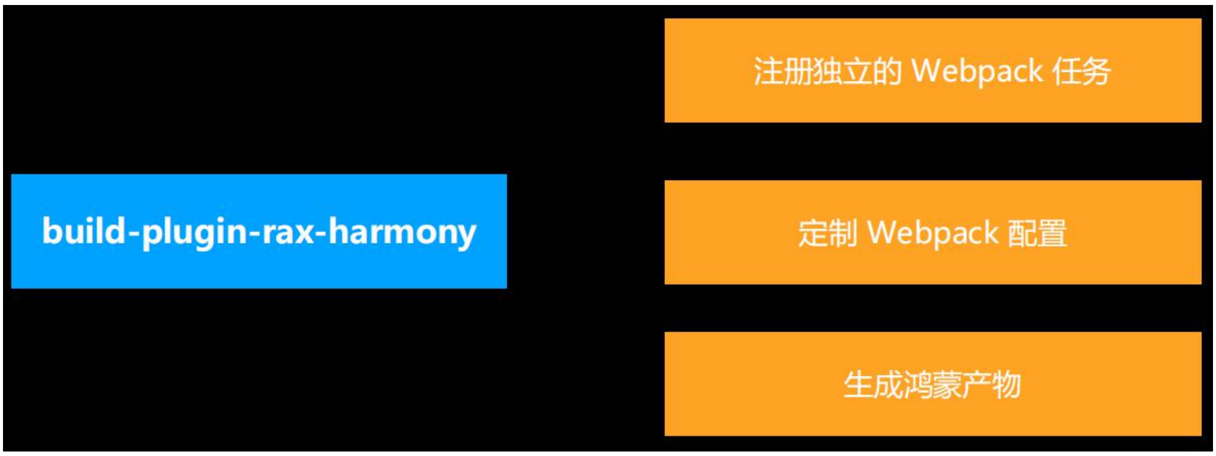
跨端方案的核心工程问题是构建多个端的代码，多端的构建可能存在差异，webpack 的配置也会有所不同，因此在 Rax 工程体系中引用了多 webpack 任务的设计。它的执行逻辑是，先加载用户配置读取 targets，注册 webpack 任务后改写 webpack 配置，最终通过 webpack 编译生成不同端的产物。



Plugin-rax-harmony

在上述基础上，就可以轻松的创建 plugin-rax-harmony 工程插件。通过这个工程插件可以：

- 注册独立的 webpack 任务为 harmony 产物的构建；
- 定制 webpack 配置；
- 生成鸿蒙产物；



3、研发链路

方案设计

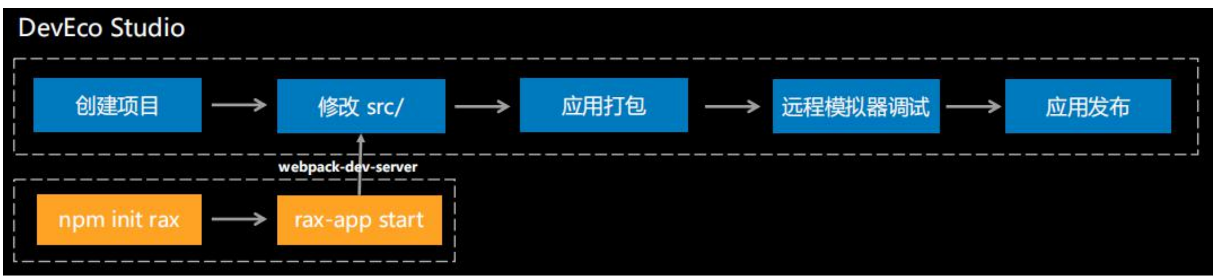
鸿蒙官方提供了 DevEco Studio IDE 支持应用的全链路开发调试与打包。



IDE 自身已经形成一个闭环，Rax 的工程链路如何对接成为核心的问题。

- 短期方案

在短期方案中会保留鸿蒙 IDE 本身提供的能力（下图蓝色框流程），同时重新创建一个 Rax 项目，生成 webpack.dev.server，将 Rax 产物打包放入鸿蒙 src 中，src 继续 IDE 链路即可。



- 长期方案

支持在 DevEco Studio 从 Rax 应用创建 -> 开发 -> 调试 -> 发布的全链路研发能力。目前这个方案还在和鸿蒙的团队在讨论之中。

（研发链路演示见视频）

附：Rax 与鸿蒙的合作 Roadmap

2021 年 8 月：Rax 团队和鸿蒙团队第一次会议交流。
2021 年 10 月：结合鸿蒙提供的渲染指令跑通 demo。
2021 年 12 月：完成渲染链路的完整对接；与鸿蒙 IDE 团队沟通全链路方案。
2022 年 2 月：IDE 全链路流程开发完成；Rax 相关组件&API 适配完成。
2022 年 3 月：结合 Harmony 3.0 正式发布。
Rax 对接鸿蒙的技术方案选择了相对性能优越、运行轻量的长期方案，预计在 2022 年 3 月结合鸿蒙 Harmony3.0 一起正式发布。

V8 JS AOT 化的探索与实践

摘要：JS 语言的动态性非常优秀，其弱类型等语言特性也使得一线业务开发者更容易上手，但这也导致 JS 每次运行前都要重复编译，使得 JS 的执行性能不理想；虽然之前 UC 内核有做过 Code Cache 方案，但支持的场景不够完整，与原生 Native 的技术方案比，尤其是首次启动场景（如各类大促活动等）还是有比较大的差距。为了能尽可能做到与 Native 对标，缩小性能差距，同时让业务开发者无感，我们开发了 JS AOT 功能。本分享将结合目前集团内自有业务形态，以及 JS 在 Web 中的执行过程，介绍 JS AOT 是如何设计和实现的，以及能给业务带来哪些收益。

分享人：喻世江，阿里巴巴高级无线开发专家，2012 年加入 UC 内核团队，参与基于 WebKit 的 U3 和基于 Chromium 的 U4 内核研发。2019 年开始专注于 V8 以及 JSI 引擎的开发优化工作，致力于将 U4 内核的 V8 引擎打造成为集团内最快、最好用的移动端 JS 引擎。

目录：

一、背景
二、技术选型
三、方案设计
四、优化效果
五、展望

一、背景

随着 Web 技术的不断发展壮大，JavaScript 作为 Web 的逻辑开发语言，也凭借其简单易用等特性以及强大的生态，成为行业内最受欢迎的编程语言之一。V8 作为 JavaScript 最强大的虚拟机引擎，在性能、稳定性、内存等各方面的指标受到业务的关注。

U4 内核在 V8 引擎中深耕多年，进行了多方面的优化和能力扩展，为了使集团业务在 Web 快速发展，消除在使用 Web 应用时的各种问题，我们 U4 内核团队从快、强、稳三个方向，在 V8 引擎中做了大量的优化和能力拓展：

1、快

- 磁盘代码缓存技术：避免 JS 非首次执行时的重复编译，使 JS 性能提升 50%以上；
- UC LLVM 优化编辑器技术：为 V8 字节码处理程序生成更高效紧凑的汇编指令，使 JS 性能提升 10%-30%；

2、强

- JS 卡死检测：当 JS 出现卡死情况时，在 native 中可以得到告警并获取卡死时的现场信息，使业务尽早发现、定位和解决问题；
- OOM 定位信息：在崩溃日志中增加了 JS 堆内存信息和 OOM 时的 JS 现场信息，使业务可以快速定位 OOM 问题；
- JS API 扩展：在 JS 中扩展了一系列能力，让 H5 开发者可以快速开发调试以及进行性能分析等；
- JSI：让业务可以脱离 WebView 使用 V8，同时享有 U4 针对 V8 的所有优化成果；

3、稳

- 疑难崩溃攻克：解决了线上发现的许多疑难崩溃稳定性问题，如 CPU 缓存刷新问题、高通部分 CPU 缺页中断处理 BUG 等系列非常底层的问题；
- 安全漏洞修复：修复各种 N-day、0-day 的安全漏洞并及时上线。

U4 内核在 V8 引擎中做了很多优化和能力扩展，然而在将 JavaScript 和传统的 Native 技术相比时，仍然发现一些不足。

如下图所示，JavaScript 在跨平台和动态化方面较 Native 有很大优势，但在启动性能方面却存在差距。其原因主要是 JavaScript 在运行前需要在用户的设备里在线编译，并且每次运行前都需要重新编译；而 Native 是在打包时 PC 离线编译，在用户设备中可以直接运行。

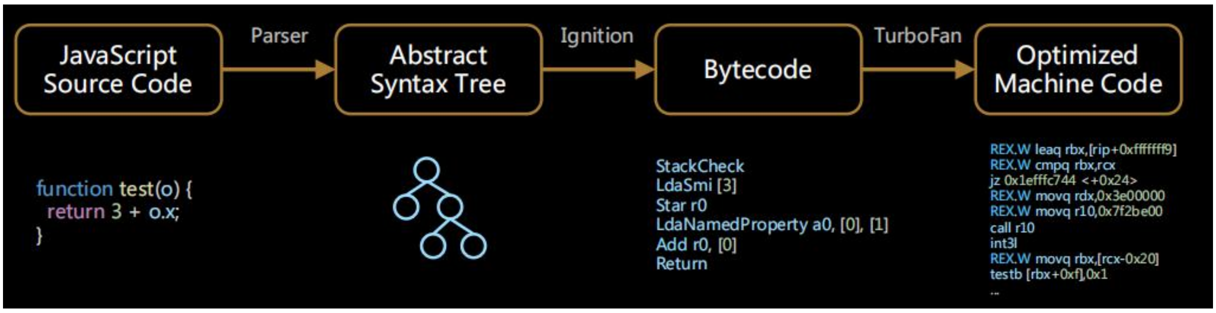


简单说 Native 用的是 AOT（Ahead of Time，提前编译）化的语言，因此，我们也尝试将 AOT 技术应用到 JS 语言中。

JS AOT 的目标是让 JavaScript 具备动态化特性的同时，运行性能也可能与 Native 对标，尤其是首次启动。

二、技术选型

下图是 V8 运行 JavaScript 的流水线。



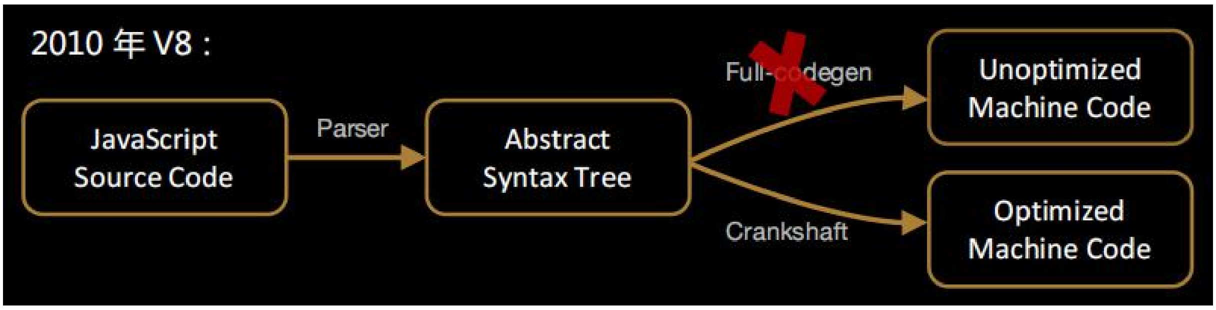
V8 运行 JavaScript 的流水线

首先 V8 会使用解析器将 JS 源代码进行词法分析、语法分析和语义分析，生成抽象语法树（AST），再使用解释器将 AST 编译生成字节码，然后以字节码的形式解释执行，在解释执行的同时 V8 会收集 JS 对象的类型信息，当某些函数多次运行变成热点函数后，V8 会使用其优化编译器对代码进行重新编译，结合之前收集的对象类型信息重新生成高效的汇编指令，并以汇编的形式运行，以获得更高的执行性能。

根据 V8 执行 JS 的流水线分析来看，实现 JS AOT 有三种可行方案：本地代码（Machine Code，汇编），全字节码缓存（Full Code Cache），部分字节码缓存（Code Cache）。

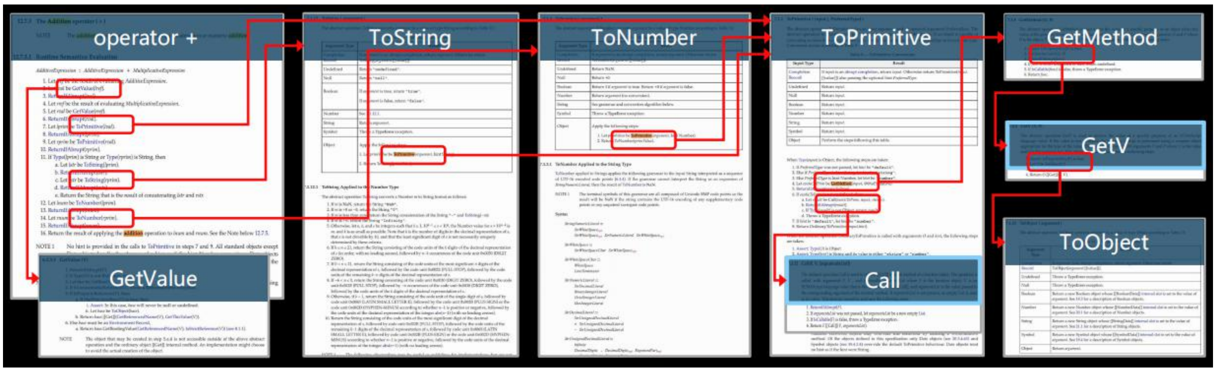
1、本地代码（Machine Code，汇编）

早在 2010 年，V8 在执行 JS 时，先使用解析器将 JavaScript 源代码解析生成抽象语法树（AST），再使用基线编译器 Full-codegen 将 AST 编译成为未优化版的汇编代码后执行；当某些函数多次执行成为热点函数后，再使用优化编译器生成优化版的汇编代码执行。然而，在 2016 年 V8 引入了字节码后，就摒弃了基线编译器 Full-codegen。



V8 为什么会摒弃基线编译器 Full-codegen 呢？这就需要对 JS 语言规范的标准有一些了解。以加号运算符为例（如下图），在 operator+的规范中，会使用到 GetValue、ToString、ToNumber、ToPrimitive 等语义，同时 ToString、ToNumber 也会调用 ToPrimitive 语义，在 ToPrimitive 语义中会调用 GetMethod、Call 语义，其中 Call 语义可能调用任意 JS 代码，GetMethod 会调用 GetV 获取对象属性，GetV 会调用 ToObject。由此可见，一个看似简单的加号运算符涉及的标准非常多，而由于 JavaScript 的弱类型语言特性，在编译时无法确定两个相加的操作数会是数值、字符串、对象等还是任意类型组合，因而需要将这些路径全部编译生成庞大的汇编代码。

通过这个示例可以看到，将 JS 源代码直接全部编译成未优化版的汇编代码，将面临编译过程缓慢、生成代码体积庞大等问题，进而同时会导致生成的代码占用内存高、因 CPU 缓存频繁失效导致性能差的问题，另外还存在 CPU 架构不通用的问题。由此可见，使用汇编代码并不是一个可取的方案，V8 验证过这个方案的不可行。

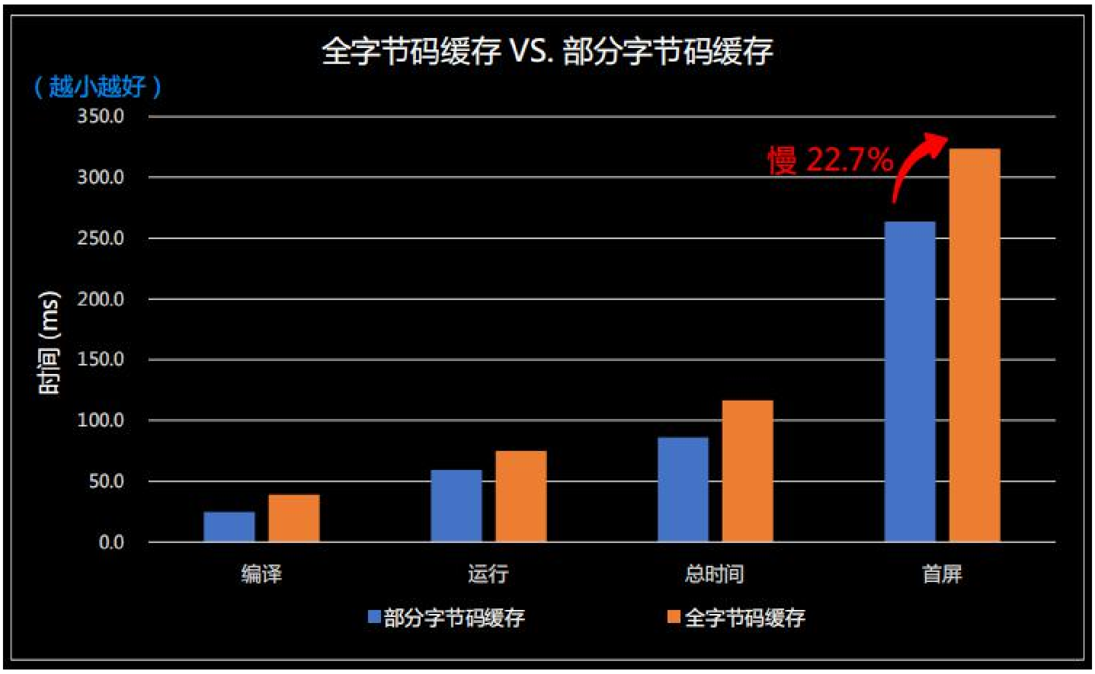


2、全字节码缓存（Full Code Cache）

V8 的字节码是一种高度抽象的二进制表示形式，每个字节码都分别对应一段字节码处理程序，每个字节码处理程序由许多条汇编指令组成。V8 的解释执行过程是按照字节码顺序依次调用字节码处理程序执行。将 JS 全部编译成字节码，以字节码的形式运行，可以有效避免运行时的编译过程，但根据线上 TOP 站点分析发现，它仍然存在一些问题：

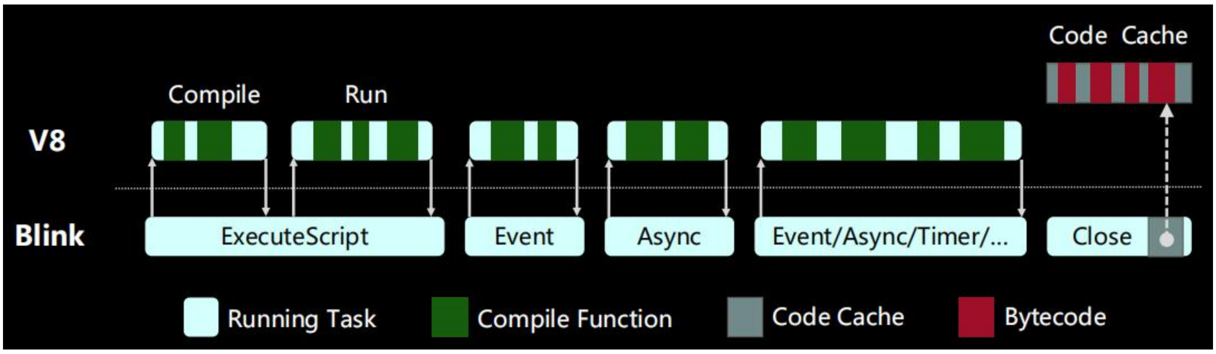
- JS 函数运行覆盖度低（运行函数个数/总函数个数=43%）；
- 代码膨胀严重（全字节码大小/JS 源码大小=2.6 倍左右）；
- 加载 & 反序列化消耗大。

根据对 TOP 站点的分析发现，对小部分 JS 来说使用全字节码缓存有较明显的性能提升，但在大部分 JS 中相比于直接编译的提升非常微小，基本可以忽略；在实际业务中，将全字节码缓存与部分字节码缓存也进行了性能对比（部分字节码缓存，即只为部分需要被执行的 JS 代码预生成字节码）。该结果表明，相比于部分字节码缓存，全字节码缓存会慢 22.7%。



3、部分字节码缓存（Code Cache）

下图是 Blink 执行 JS 的逻辑。



在 Blink 中执行 JS 时 V8 会先只编译最上层的函数，并在运行到内部函数时，如未被编译，则先编译该内部函数后运行。同时 JS 中会注册 DOM 事件，在 DOM 事件被触发时也会执行之前未执行的函数，这时 V8 也会先将其编译生成字节码再运行，同样如 Async function、Promise、Timer 等异步任务，也会在遇到未被编译的函数时先编译再运行。最后，当页面跳转或关闭时，将所有编译过的函数序列化并生成字节码缓存。

这个方案存在的问题：

- V8（U4 内核）版本的碎片化：U4 内核对 V8 深入优化修改时，可能改动到 V8 内部的数据结构，导致不同版本不通用；
- 不同 CPU 架构下的通用性：V8 的字节码是全平台、所有架构一致通用的，但是字节码所使用的部分常量数据（如小整数）的内存布局在不同架构下不一样，导致整个字节码缓存与 CPU 架构相关；
- 机型通用性：高、低端机上生成的字节码缓存可能不通用。

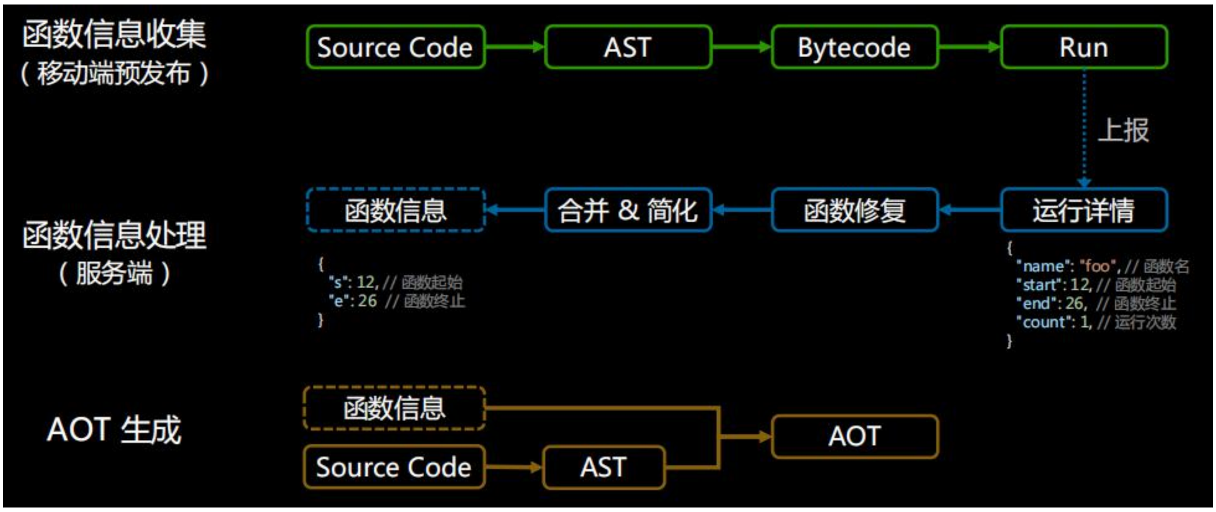
根据以上三个方案总结来看，JS AOT 面临的主要问题在于生成代码的有效性和兼容性。

三、方案设计

1、保证 AOT 的有效性

策略一：追求极致的性能——PGO（Profile-guided optimization）

PGO 是基于性能测试结果的优化，JS 使用 PGO 的思路是仅为需要被执行的函数生成代码，整个流程如下图。

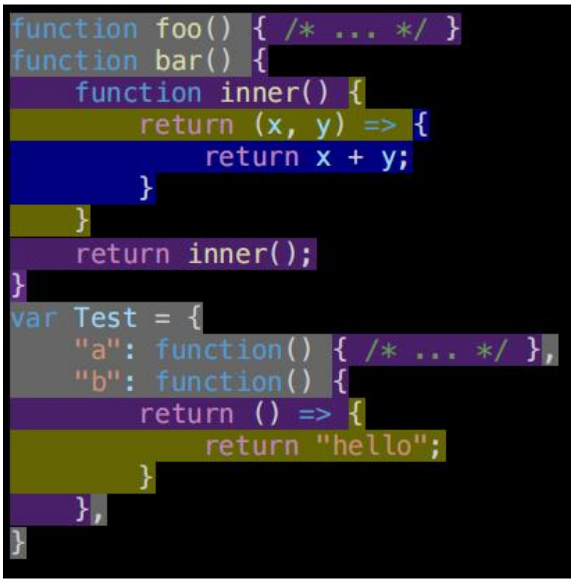


在业务发布之前先在移动端访问一次页面，JS 执行完后会生成函数运行信息并上报给服务端，服务端结合 JS 源码对信息进行处理，如补回已被 GC 的函数、合并和简化信息等，最后在服务端或移动端使用函数信息结合 JS 源码预编译生成 AOT 代码。

策略二：追求便捷的使用——先验规则

使用 PGO 的方式生成 AOT，每次发布时都需要先收集函数信息，这在无形中增加了业务发布的复杂度。为了追求更便捷的使用，内置了另一种先验规则的策略。这个策略的核心在于预测需要被执行的函数，然后只为这些函数生成字节码。

下图是一段 JS 代码，在代码执行前 V8 先编译灰色背景部分的代码，这些代码运行时会编译并执行第二层代码（紫色部分），第二层代码执行时会编译第三层代码（黄色部分），然后是第四层代码（蓝色部分）。



通过对线上 TOP 站点的 JS 分析，发现了以下特征：

- JS 层级嵌套越深，使用率越低；相反，层级越浅使用率越高，如最上层代码的使用率是 100%；
- JS 越小，覆盖度越高：一些小型 JS 内的函数使用率可以达到 90% 以上。

结合以上特征对 TOP 站点进行不断的实验和分析后，最终确定了一套整体最优的生成策略，如下：

- 小型 JS：不生成 AOT

因为如果生成 AOT 后都需要从磁盘加载反序列化，可能会比直接编译 JS 源码更耗时。

- 中型 JS：全字节码缓存

因为这部分 JS 函数覆盖度达 80% 以上，使用全字节码性价比较高。

- 大型 JS：只生成 Top 3 层函数字节码

其它未被编译的函数在运行时用到会及时编译。

- 运行后增量更新

页面生命周期结束时，将新增编译的函数序列化，增量更新保存；运行次数越多，AOT 包含的函数就越多，运行性能越优。

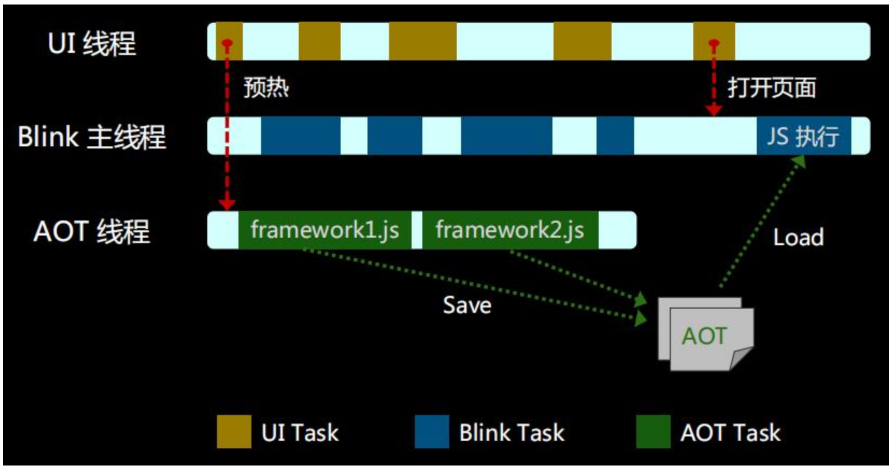
2、保证 AOT 的兼容性

策略一：在线生成

空闲时预热

下图是简化版 U4 内核线程模型图。首先，在 UI 线程发起 AOT 预热，然后在 AOT 线程将 JS 预编译生成 AOT 缓存在磁盘，当用户打开页面，Blink 主线程在运行 JS 代码时加载 AOT，避免了 JS 的编译过程从而提升性能。

- 影响或不足：存在资源浪费（磁盘 & CPU），部分用户可能不会访问相应的页面；
- 适用场景：框架 JS，不经常变动。

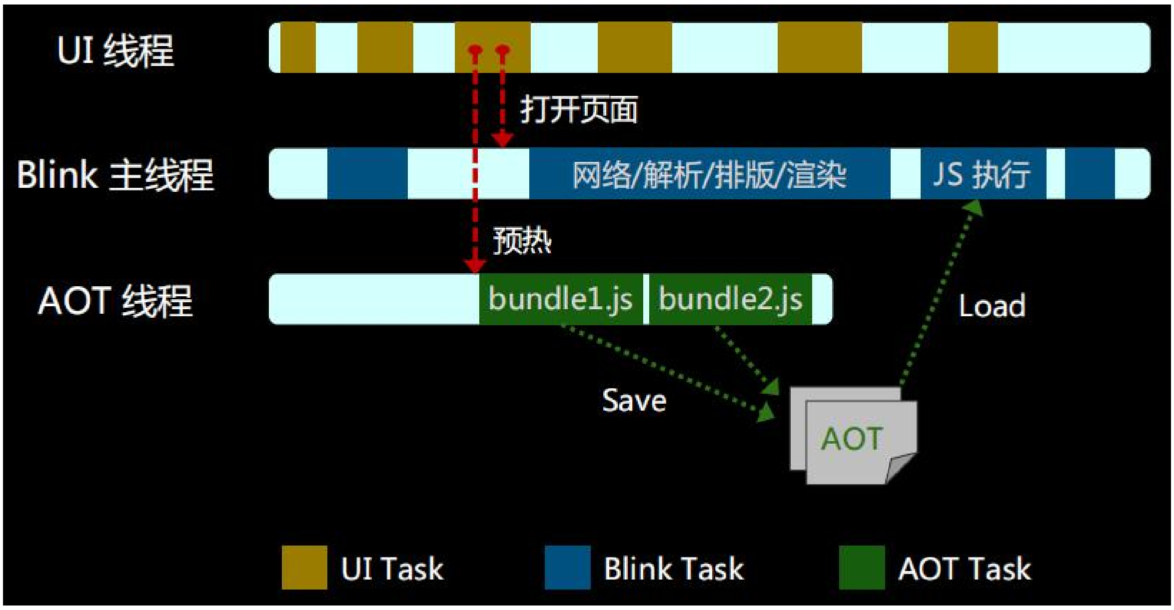


在线生成（空闲时预热）

访问时生成

如下图，当用户打开页面时向后台 AOT 线程发起预编译任务，让后台线程同步生成 AOT，同时 Blink 主线程发起页面打开加载请求，并进行解析、排版、渲染等操作，在结束这些操作时 JS 代码也已经在 AOT 线程编译完成，Blink 主线程可以直接加载和反序列化并运行。

- 影响或不足：如果 AOT 线程执行缓慢，在 Blink 主线程执行 JS 时字节码可能未生成完成，导致 Blink 主线程需要重新编译；
- 适用场景：业务 JS，已经离线到本地。



在线生成（访问时生成）

策略二、离线生成

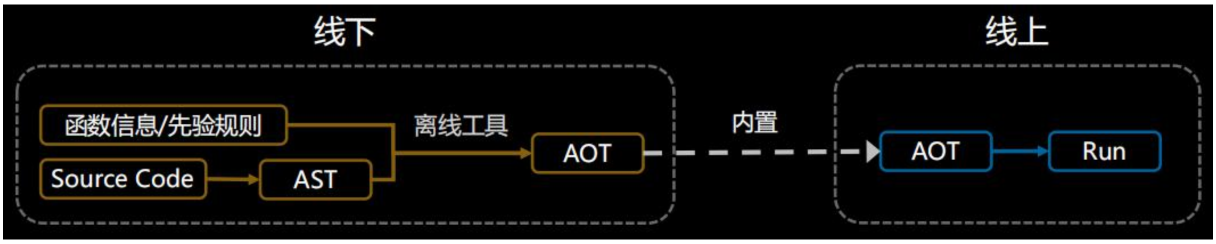
离线生成的方式提供了一个离线工具，可以在服务端使用函数信息或先验规则结合 JS 源码预编译生成 AOT，然后将 AOT 内置到发布包中，提供给线上用户直接运行，省去编译的过程。

适用场景：

- APP 冷启动时执行（没有预热时机）；
- JS 不经常变动；
- JS 不需要动态更新。

影响或不足：

- 更新 U4 内核后，可能需同步更新 AOT，如果不更新，可能导致加载失败，需要运行时重新编译。



离线生成

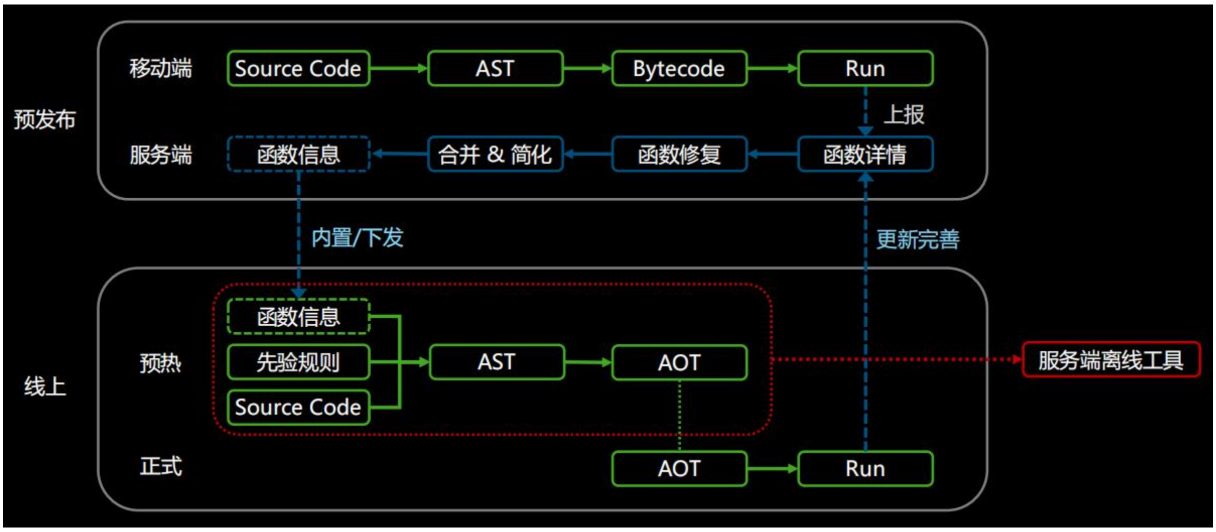
既然使用了 AOT 离线生成，那是否可以在线上只保留 AOT 而抛弃 JS 源码吗？答案是不可以，原因有以下两点：

- 抛弃 JS 源码意味着需要将 JS 全部编译成字节码，前面提到的 PGO 或先验规则，都只是生成部分函数的字节码，而有些 JS 代码是在非主路径才会使用，如果没有 JS 源码，在调用这些函数时就会出现异常；而使用全字节码缓存则会出现前面提到的问题；
- JS 的一个特殊功能是将任意函数通过 toString 调用取得原始的字符串，如果将 JS 源码抛弃，这个功能就无法实现。

3、AOT 方案总览

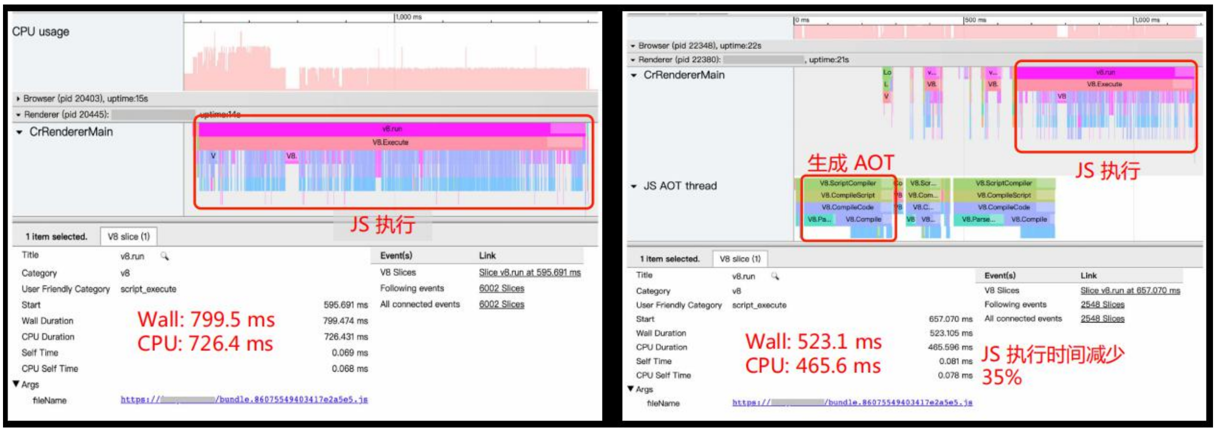
根据以上介绍的所有内容，AOT 整体方案总结如下：

在预发布环节，首先在移动端访问页面，然后上报函数信息至服务端，服务端进行信息处理后生成函数信息（如果使用先验规则策略则可省去预发布环节）；在线上后台线程中使用函数信息或先验规则结合 JS 源码进行预编译生成 AOT，当用户访问页面时可以直接从 AOT 运行而无需编译，从而提升 JS 执行性能；对于预热环节，还提供了离线工具，可以在服务端离线生成 AOT。



四、优化效果

下面左图是某页面未使用 AOT 的 Trace 结果，右图是基于先验规则打开页面时后台预热生成 AOT 的结果。对比两个结果发现，后者 JS 执行时间减少 35% 左右。



另外，针对线上 TOP 站点进行对比分析发现，使用 PGO 策略可以使执行性能平均提升 49%以上，使用先验规则可以平均提升 33.9% 以上。我们以夸克高考为例，对比使用 AOT 后，首屏性能提升了 17.6% 以上。

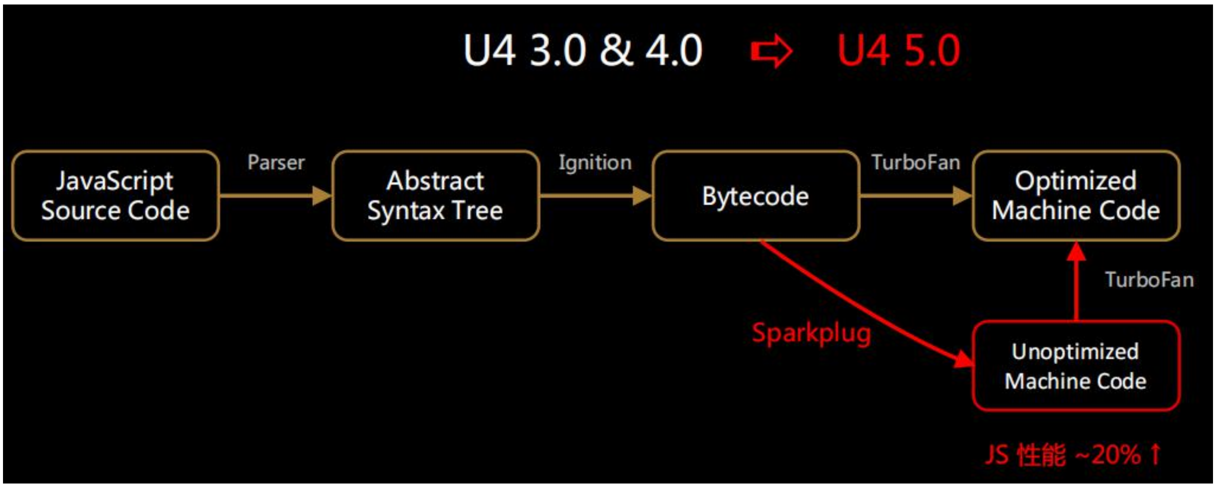
五、展望

在当前的 U4 3.0 & 4.0 版本中，V8 执行 JS 的流程是：通过解析器将 JS 源码解析生成抽象语

法树，然后使用 Ignition 解释器将抽象语法树编译成字节码并以字节码形式运行；当部分函数多次运行变成热点函数后，V8 使用优化编译器将其编译成优化汇编代码并运行。

而在即将发布的升级 U4 5.0 版本中，使用 V8 的火花塞基线编译器，可以从字节码直接编译生成未优化版的汇编代码并运行。当部分 JS 代码多次运行成为热点函数后，会再次使用优化编译器重新编译，生成优化版的汇编代码。

新增的火花塞基线编译器，在保留字节码抽象度的同时，将字节码的解码过程提前，并能够充分利用 CPU 的分支预测能力，从而提升 JS 的执行性能。初步预测，如果 JS AOT 使用火花塞编译器后，我们在 AOT 中可以进行更多的优化，使 JS 性能提升 20% 左右。



以上。感兴趣的朋友欢迎关注 U4 内核技术公众号了解更多内容。

浅谈 web component 实践与应用

摘要：前端框架不断推陈出新，如何在不断的革命与淘汰的氛围中维持服务功能的稳健，一直是每位工程师最大的理想与目标，本片分享的是如何透过 web component 轻松快活玩转各框架，让功能开发与服务稳健可以并行迈向未来。

分享人：Paul Li，台湾雅虎 AMP 项目的首席程序员，一直渴望使用现代网络技术。 他也是台湾雅虎的前端工程师，专注于用户体验，为用户带来生动流畅的流程。

目录：

一、为什么使用 Web Component?
二、Custom Elements（自定义元素）
三、Shadow DOM
四、实践演示

八年前，Web Component 推出后即受到 Google 的青睐，旗下浏览器率先支持 Web Component。但由于当时环境并不友好，同时 Web API 的支持不够充足，Web Component 没有得到推广应用，然而它的核心思维启发了近年来 Framework 的开发。

经过几年的发展，Web Component 已经成功的成为 W3C 的标准之一，主流浏览器如 Chrome、Edge、Firefox、Safari 都纷纷张开双臂拥抱 Web Component。

一、为什么使用 Web Component?

1、背景

雅虎奇摩拍卖是一个类似于淘宝的电商平台，在台湾服务近 20 年，深受大家喜爱。然而由于技术转换、人力缺乏等一系列原因造成的技术问题，导致无论是既有功能的移转，还是新功能的开发，工程师都需要以不同的 Framework 开发相同的 module，这无形中增加了工程师的工作时间，同时削弱了他们的工作热情。然而这些问题在导入 Web Component 后都得到确切有效

的解决。

2、Web Component 的价值

Web Component 的使用非常简单便捷，当需要什么功能时，只要置入对应的 Web Component 就可以完成功能的导入，其价值主要体现在以下三点：

- 跨 Framework 独活：使用 Web Component 不用担心与不同 Framework 页面相冲突，而且它可以快速转换到不同页面，十分灵活；
- 操作简单便捷：让工程师重拾开发的喜悦，同时感受如玩乐高一样的乐趣；
- 重复使用性高，One for All：编写好一个 Web Component 可以快速将其使用到不同的页面不同的环境。

二、Custom Elements

1、Custom Elements 是什么？

通过 Custom Elements，开发者可以快速定义需要的 HTML tag，并将一些功能封装。比如：将上传图片封装到特定的 HTML tag，当有上传图片需求时，可以直接调用相应的 HTML tag，即可实现此功能。另外，Custom Elements 还可以监听 attribute 和 property 的变化，并进行相应的处理。

2、如何设定 Custom Elements？

通过 JavaScript class 撰写，首先定义名称：MyComponent，让它直接扩展 HTMLElement，然后通过 Custom Elements 的反馈在不同时期不同时间点的 callback 操作。当 connectedCallback 时可以进行 Event 监听的一些附加，当 disconnectedCallback 时就可以将 Event 附加进行 issue 操作。

如前面提到，Custom Elements 可以支持 attributeChanged，因此当 attribute 有新增、修改或移除时就会调用 attributeChangedCallback。

如果需要 Custom Elements 支持不同的 Property，可以通过 get 和 set 设定 Property。

最后也是最重要的步骤是要通过 customElements.define，将刚刚撰写的 class 以及要定义的 tag 名称‘my-component’进行绑定，tag 名称必须包含“-”符号。完成绑定后，“MyComponent”的 HTML tag 就产生了。

```
// my-component.js
export class MyComponent extends HTMLElement {
  constructor() {
    super();
    ...
  }

  connectedCallback() {
    ...
  }

  disconnectedCallback() {
    ...
  }

  attributeChangedCallback(attrName, oldValue, newValue) {
    ...
  }

  get myProperty() {
    ...
  }

  set myProperty() {
    ...
  }
}

window.customElements.define('my-component', MyComponent);
```

3、如何使用 Custom Elements

使用 Custom Elements 只要导入对应的 JavaScript，建议使用 JavaScript Module 的方式加载档案，因为通过 JS Module 可以使用 import/export 语法快速导入 JS Lab，相同的 JS 档案不会重复加载，同时加载的 JS 可以得到保护。最后将新建的 my-component 置入到相应的地方即可使用 Custom Elements。

```
window.customElements.define('my-component', MyComponent);


// HTML
<script type="module" src="my-component.js"></script>
<my-component></my-component>


// JavaScript Instantiation
<script type="module">
import { MyComponent } from './my-component.js';

// use DOM API
const nodeA = document.createElement('my-component');
document.body.appendChild(nodeA);
nodeA.myProperty = ...;
```

另外，还可以通过 JavaScript Instantiation 的方式导入，同样使用 JavaScript Module。其它方式还包括：DOM API、new instant 等。开发人员可以根据自身环境快速选择一个最适合的方式。

```
// JavaScript Instantiation
<script type="module">
import { MyComponent } from './my-component.js';

// use DOM API
const nodeA = document.createElement('my-component');
document.body.appendChild(nodeA);
nodeA.myProperty = ...;


// new instance with Class
const nodeB = new MyComponent();
document.body.appendChild(nodeB);
nodeB.myProperty = ...;


// new instance with Class & default config
const config = {
  ...
};
const nodeC = new MyComponent(config);
document.body.appendChild(nodeC);
</script>
```

（详见视频演示）

三、Shadow DOM

通过 Shadow DOM 可以使 Web Component 呈现出表里不一的效果。如果 Web Component 同时具有 Light DOM 和 Shadow DOM 时，浏览器会以 Shadow DOM 为主要渲染对象。使用 Shadow DOM 旨在保证 Light DOM 和 Shadow DOM 的渲染不会互相干扰。

如何将 Shadow DOM 与 Custom Elements 有效结合?

打开刚建立好的 my-component.js，首先生成一个 template，设置 innerHTML，设置的内容就是希望 Web Component 呈现的内容。接着在 constructor 里设置 Shadow DOM，然后将 template 里的内容置入 Shadow DOM，实现 Shadow DOM 和 Custom Elements 的结合。

```
// my-component.js
const template = document.createElement('template');
template.innerHTML = `
<style>
...
...
</style>

<aside>
...
</aside>
`;

export class MyComponent extends HTMLElement {
  constructor() {
    // template
    this.attachShadow({ mode: 'open', delegatesFocus: true });
    this.shadowRoot.appendChild(template.content.cloneNode(true));
    ...
  }
  ...

}

window.customElements.define('my-component', MyComponent);
```

Web Component 的 type 名称都是由开发者自定义，因此搜索引擎无法第一时间判断其含义，这时可以通过 Light DOM 不会被渲染的特性设置一些线索提供给搜索引擎进行分析，以确保 SEO 的准确性。

```
// HTML
<script type="module" src="my-component.js"></script>
<my-component>
  <h3>my-webcomponent</h3>
  <p>
    content in Light DOM won't render.
  </p>
  <p>
    We could put something here for SEO.
  </p>
</my-component>
```

另外，还可以基于 CSS 完成对 Style 的附加，比如“:host”是 Shadow DOM 搭配“:hover”实现样式的变更等，见下图。

通过以上操作让 Web Compoment 更加灵活，以满足不同的需求，实现不同的呈现效果。

```
// Change looking
const template = document.createElement('template');
template.innerHTML = `
<style>
:host(:hover) {
  ...
}
:host([disabled]) {
  ...
}
:host(.blue) {
  ...
}
:host {
  --gap: 1px;
  --color: #fff;
  --border-radius: 8px;
}
</style>
...
`;
```

四、实践演示

以雅虎奇摩拍卖为例，介绍 Web Component 在多元场景的应用。

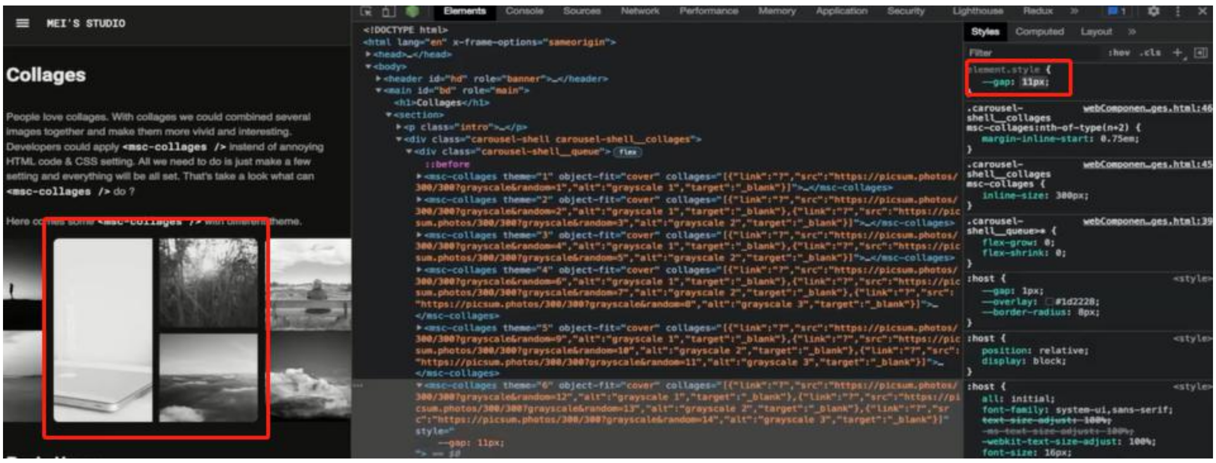
1、Collages

基于 Web Component 可以实现不同服务中呈现不同图片的拼接效果。





举例：如下图，如果要调整红框内图片的间距，可以通过改变 gap 设置实现。



2、EZ Video

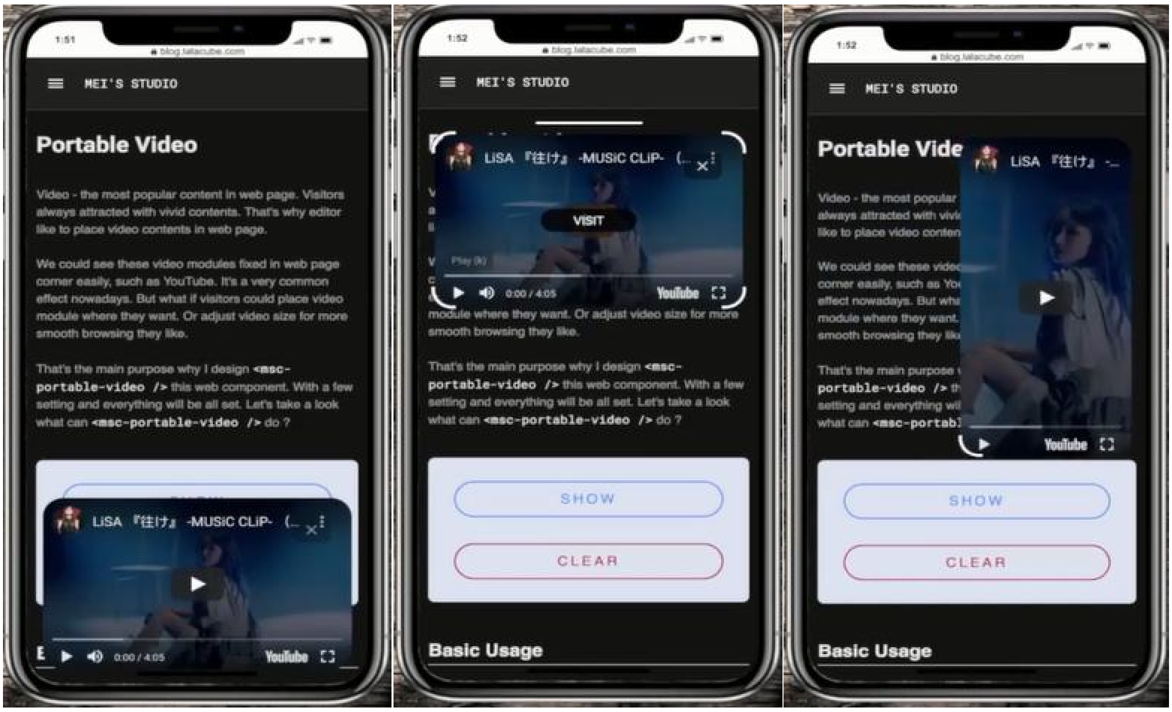
雅虎的程序开发者和设计师希望视频可以呈现雅虎特有的元素，EZ Video 就是专为视频设计的 Web Component。其中包含常见的视频播放能力如：快进快退、静音、画中画、全屏，等等都被封装入 Web Component 中。





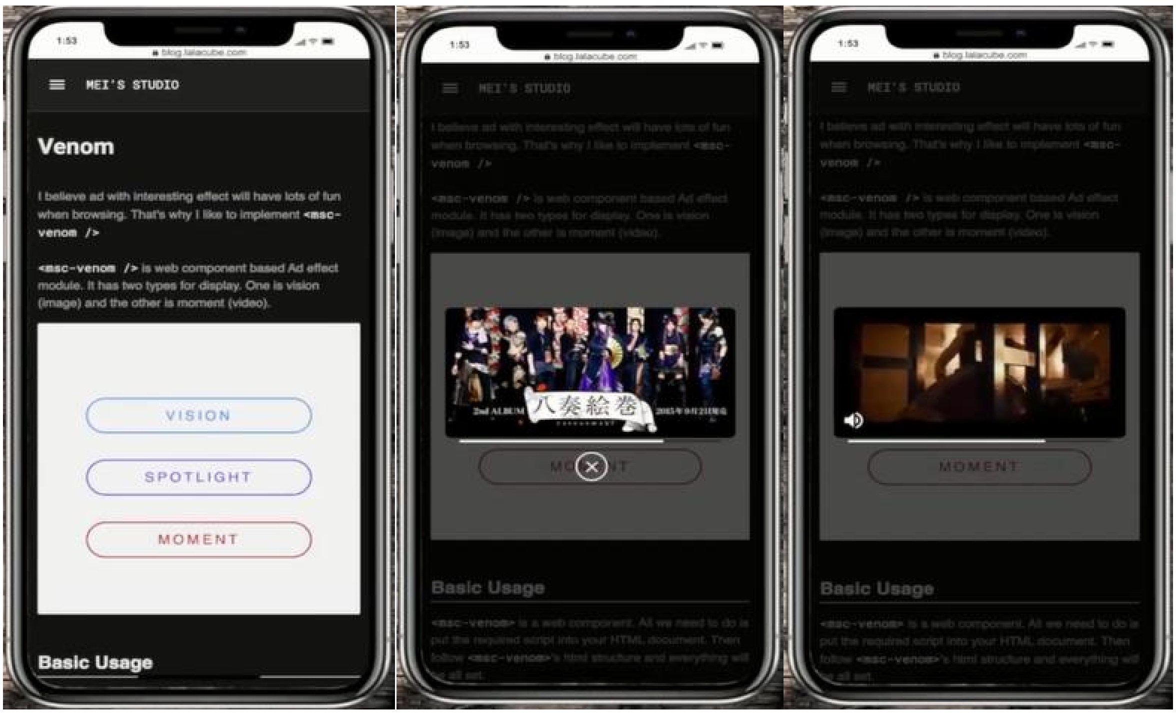
3、Portable Video

随着大陆直播带货的风潮登陆台湾，雅虎奇摩拍卖也借鉴了这种营销形式。业务希望将直播的窗口在不同页面推广，让更多人有机会看到。基于这个需求开发了 Portable Video，它不但可以自由放置在不同页面，而且在同一页面用户还可以随意变换视频的位置和大小，实现更加舒适的页面浏览体验。



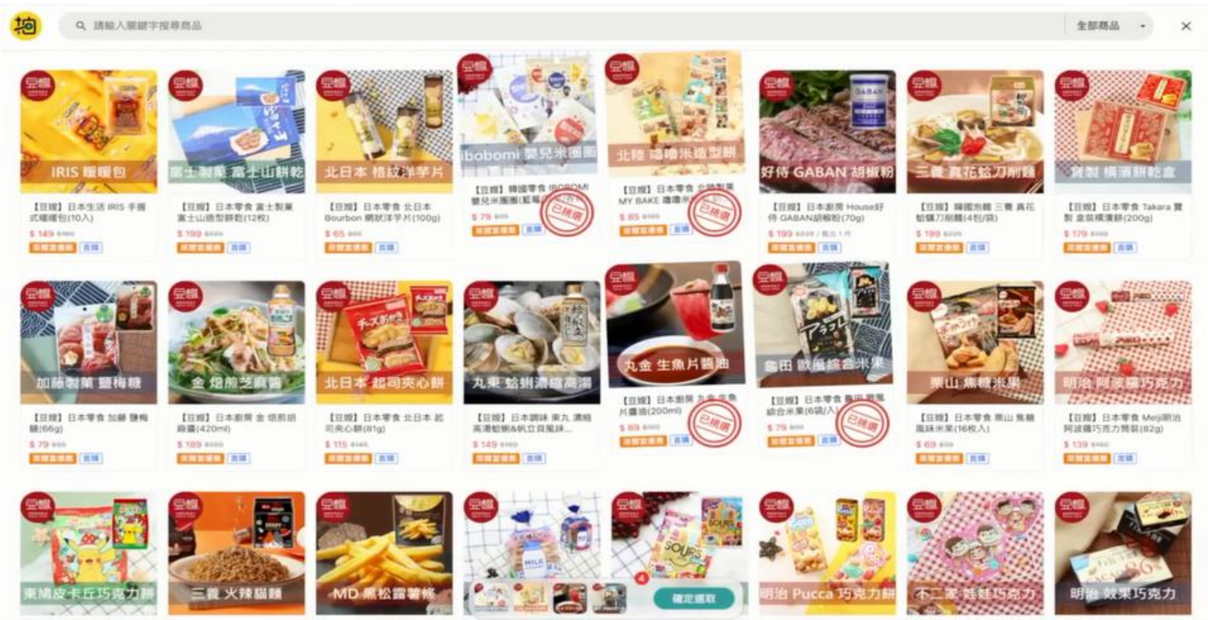
4、Venom

Venom 就是雅虎开发的广告封装 Web Component，有图片、video 等不同呈现方式。通过 Web Component 包装广告让广告效果得到更好的发挥。



5、Product Picker

选品是平台商家从陈列产品中挑选希望售卖的产品。Product Picker 已经将产品相关信息包装到 Web Component，商家只需简单设置 config 就可以实现选品功能。



6、雅虎奇摩拍卖的其它 Web Components 应用

示例 1：

有些摩主让卖家自己定义 HTML 和 style，这一定程度上会形成对页面整体 style 的影响。针对这一问题，通过 Web Component 的撰写将其封印，使 Web Component 内的 style 与外部页面 style 互不影响。

示例 2：

如下图右下角的提示框也是一个 Web Component，如商家需要可以快速导入到自己的页面。



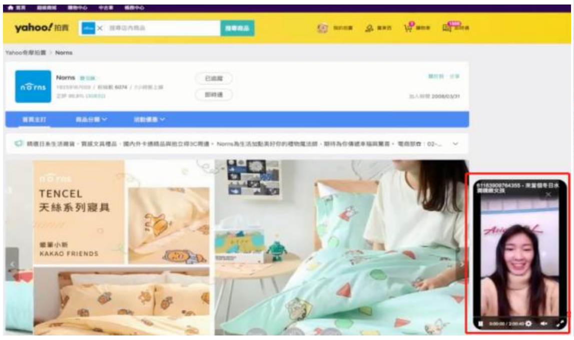
示例 3：

在卖家产品上传页面中的图片上传也可以封装为 Web Component。



示例 4：直播带货

用户可以在页面的任何位置放置直播视频，并且可以在浏览页面的同时观看直播。除了雅虎奇摩拍卖，直播视频还可以顺利导入到不同 Framework 的服务页面，如雅虎购物中心、雅虎商城，甚至在淘宝页面也可以实现。


TENCEL
天絲系列寢具


免運
優惠活動


滿額最高折$360
募集你的
FREE


品牌日
1折起
時時樂


淘宝
Taobao
中国
大陆
台灣¥249包郵
跨境購物0運費
百萬件商品 可跨店任選
立馬逛逛
品牌折扣马上抢
时尚新品 潮流好货
¥558
¥279

Serverless

云原生 WebAssembly 与 Serverless

摘要：WasmEdge runtime 是一个轻量级高性能 WebAssembly 虚拟机，特别为云原生和边缘优化。WasmEdge 最近被 CNCF 基金会接受为沙箱项目，这是全球范围内首个被 CNCF 纳入托管的开源 WebAssembly runtime 项目。Docker 的创始人 Solomon Hykes 曾提到过，如果 WASM+WASI 在 2008 年存在，他们将不会创造 Docker，足以说明 WebAssembly 在云原生领域带给人们的想象空间。本次分享由项目创始人 Michael Yuan 博士带来，与大家分享 WebAssembly 在云原生的发展趋势、应用前景以及前沿创新，值得期待。

分享人：Michael Yuan 博士是 WasmEdge 的创始人和核心开发者。WasmEdge 是一个由 CNCF 托管的开源 WebAssembly 虚拟机。其主要应用于边缘计算、Service Mesh 和嵌入式函数。 Michael 撰写过 6 本软件工程书籍，也是长期的开源贡献者。 Michael 同时也是 Second State 的联合创始人。Second State 是一家开源基础软件初创公司，致力于商业化可持续地支持 WebAssembly 和 Rust 生态系统中的企业应用。

目录：

一、WebAssembly：全站轻量级的容器
二、云原生 WebAssembly
三、WebAssembly 生态的创新

最近很火的一个新闻，Adobe Photoshop 可以在浏览器里运行起来了，这还是怎么做到的呢？Adobe Photoshop 的大部分程序都是用 C、C++写的，Adobe 把 C、C++的库编译成 WebAssembly，就可以在了浏览器里边运行起来。目前现有的程序都是用 C、C++程序编写的，能够把 C、C++的程序编译成在浏览器里边运行的一种程序，一种格式，对于前端开发来讲是一件很好的事儿。

现在前端的开发越来复杂，前端的开发者越来越不满足于传统前端的技术需求，开发者越来越愿意有自己的后端（服务于前端的后端 Backend For Frontend 即 BFF），WebAssembly 在 serverLess 里边是一个容器取代的好选择。

一、WebAssembly：全栈的轻量级容器

WebAssembly 是一个全栈的轻量级容器，不仅是一个虚拟机，而且能够提供安全性、可调度性。能够提供内存的安全，也能够提供网络的安全。2020-2021 年 WebAssembly 在容器行业走到了前沿。

1、WebAssembly 的迁移历史

- 作为浏览器沙盒被提出，目的是增加 JavaScript 的性能；
- 2019 年，被 W3C 认证为浏览器的第四大语言；
- 同年，WebAssembly System Interface 提出，赋予了 WebAssembly 访问系统的功能；
- 服务端的 WebAssembly 走在了前沿：
  - Bytecode Alliance 成立；
  - CNCF 接收了三个 WebAssembly 项目；

2、WebAssembly vs Docker

WebAssembly 能否取代 Docker?

WebAssembly 定义为容器，不可避免的要进行对比。2019 年 Docker 这个团队认为，WebAssembly 和 Docker 是有很大的可比性的。



查看完整对比：https://wasmedge.org/wasm_linux_container/

WebAssembly 与 Docker 的相似之处

- runtime 隔离

包括安全的隔离和资源进行调度分配。Docker 的安全性是有的，但是远远不足，WebAssembly 一开始就是为公共开放环境的所设计的，WebAssembly 的一个大的应用场景就是区块链；

- 容易部署

是指获取启动分配地址网络，就是干这种底层部署自动化 WebAssembly 可以做同样的事儿；

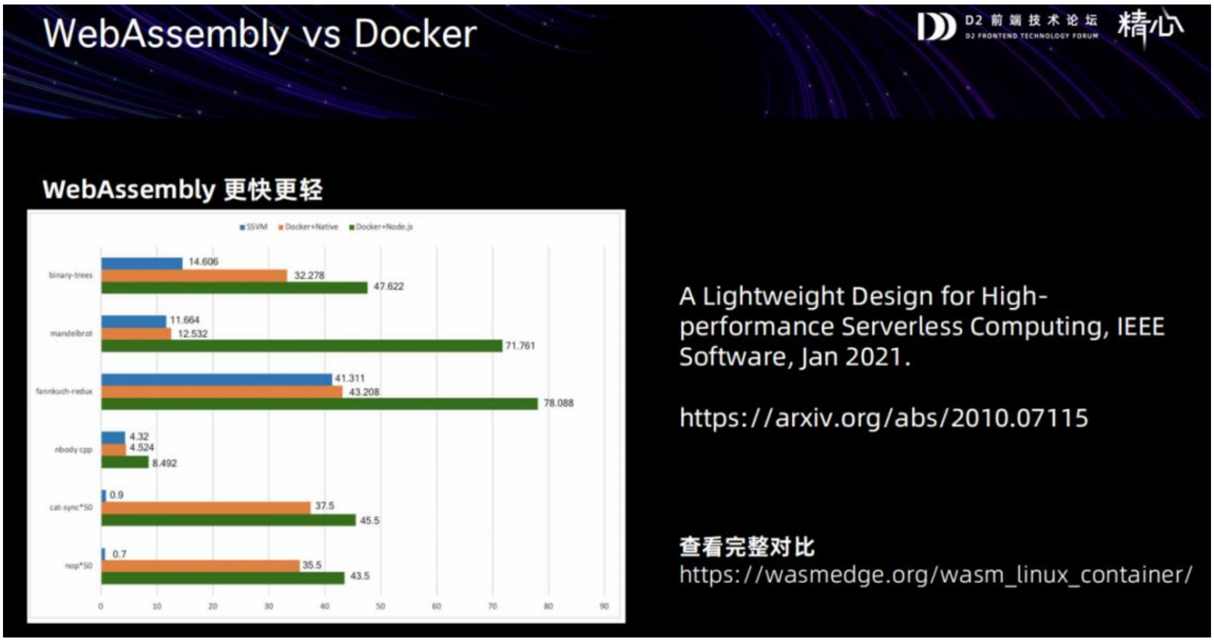
- 可移植性

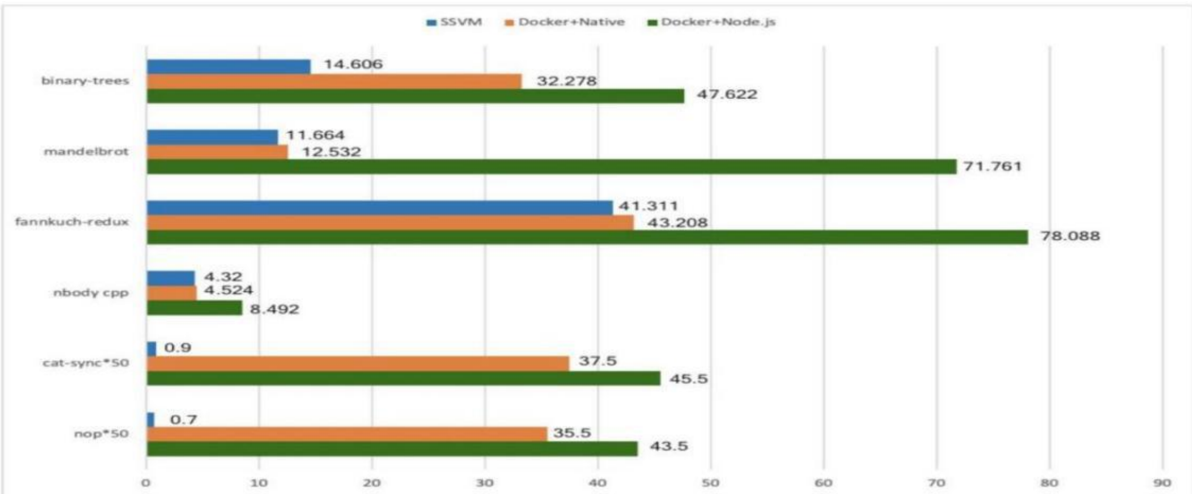
在软件层面都可以，在硬件方面 Docker 是不行的，这点是 Docke 一直想做的。WebAssembly 在软硬件各个操作平台上都可以运行。

可移植和易部署两个是相关的，不需要考虑底层平台上的硬件和操作系统的部署就容易很多，如果需要去考虑的话，逻辑分支就会有很多。所以说，WebAssembly 继承了 Docker 的这些优势，并对他进行了发展。

查看完整对比：https://wasmedge.org/wasm_linux_container/

WebAssembly 更快更轻





WebAssembly 和 Docker 的很大的不同是性能。Docker 需要模拟整个操作系统处理，而 WebAssembly 不需要，就是一个进程，只要运行起来就可以。在启动方面 WebAssembly 比 Docker 快百倍，占用资源上也小近乎百倍，运行上也快近 30%，设计初衷就是要在浏览器里边运行起来，而且速度要快很多。

查看完整对比：https://wasmedge.org/wasm_linux_container/

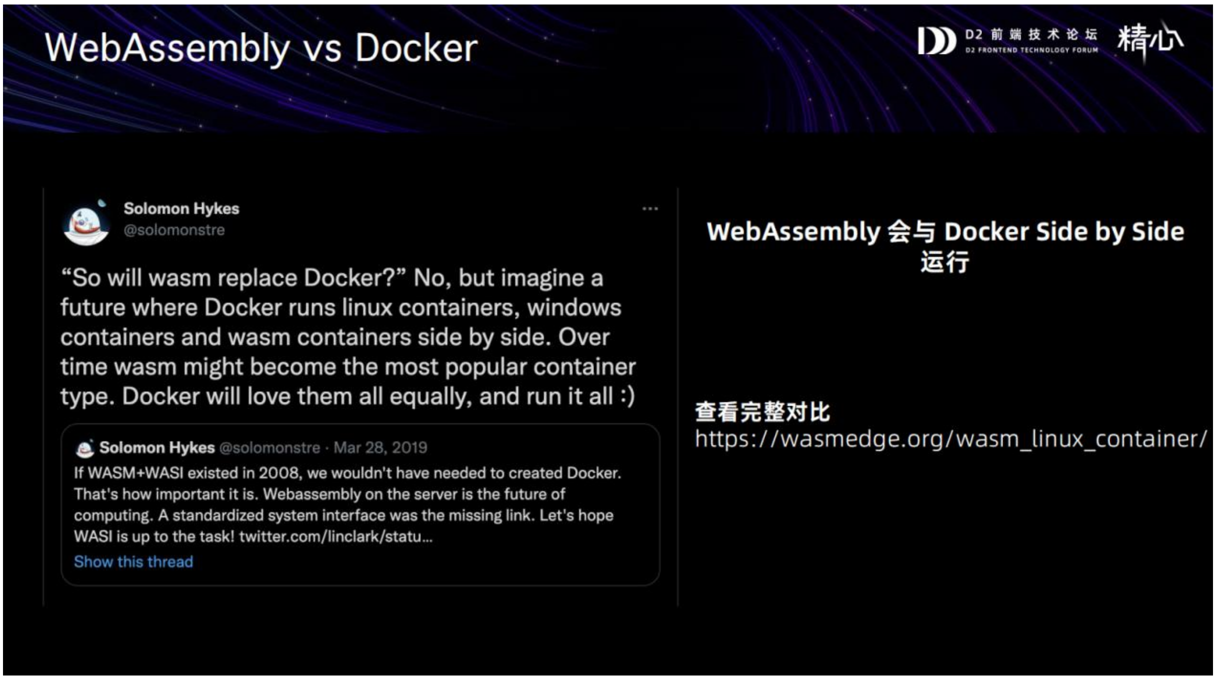
参考资料：A Lightweight Design for High-performance Serverless Computing, IEEE Software, Jan 2021.
https://arxiv.org/abs/2010.07115

WebAssembly 比 Docker 更抽象

WebAssembly 能做到这些，是因为它很抽象。

- 在云计算方面，我们的抽象层次是经历了几个阶段发展的。第一个在云计算刚开始的时候，我们都用虚拟机，Hypervisor VM 和 microVM 都是在帮助模拟一个计算机，装操作系统后再运行；
- 应用容器 Linux container，其实就是在操作系统上再模拟一个操作系统出来，需要把底层的基础库都给替换掉，这就是以 Docker 为代表的应用容器；
- WebAssembly 就是一个抽象层次更高的进程，是一个高级语言的 VM，WebAssembly 必须先编译成 WebAssembly VM，然后再用去执行，就是一个进程，可以在各种操作系统上运行，更高的层次，更好的性能，更容易管理，更容易跨平台。

WebAssembly 会与 Docker Side by Side 运行



WASM 真的会代替 Docker 吗？社区人员想到的是这种不同的容器是可以肩并肩的。Linux container、Windows container，Wasm container，里边都可以运行，进入之后，路由到不同的 container 里，用不同的 application 来处理它，这是我们看得见的未来。有轻量级的容器，也有重量级的容器。

查看完整对比：https://wasmedge.org/wasm_linux_container/

WasmEdge



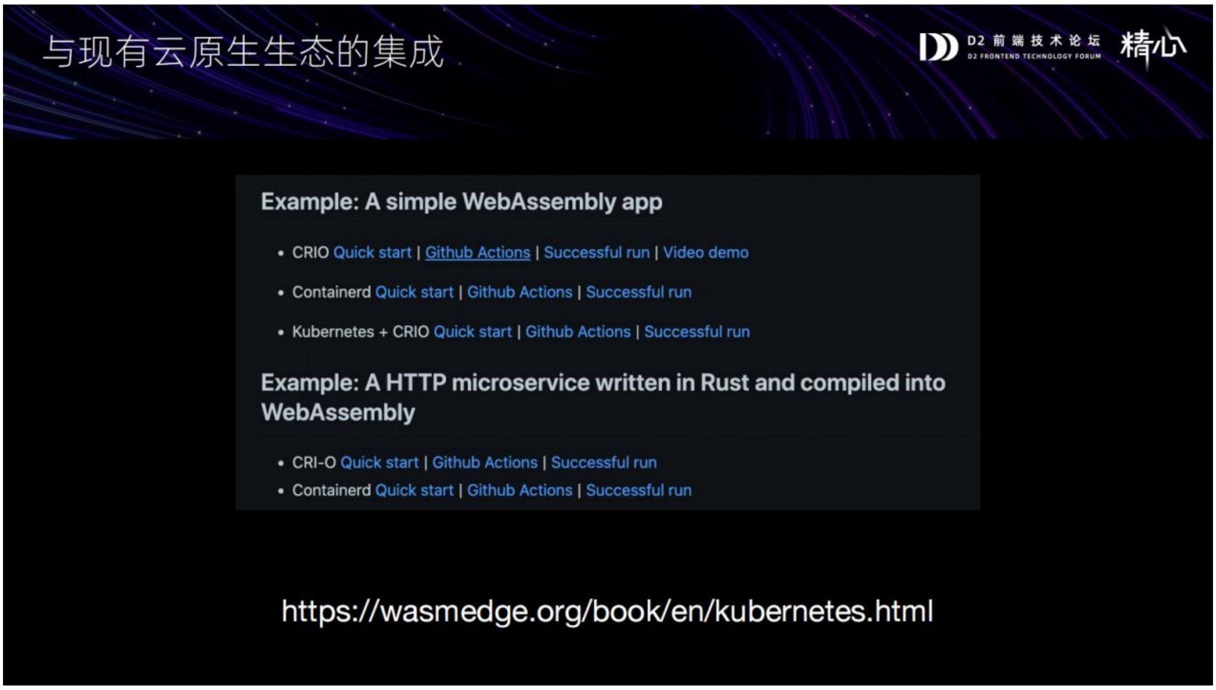
WasmEdge 是为云原生和边缘优化的 WebAssembly Runtime，是 CNCF 的沙箱项目，欢迎大家参与进来我们的这个项目中。

https://github.com/WasmEdge/WasmEdge

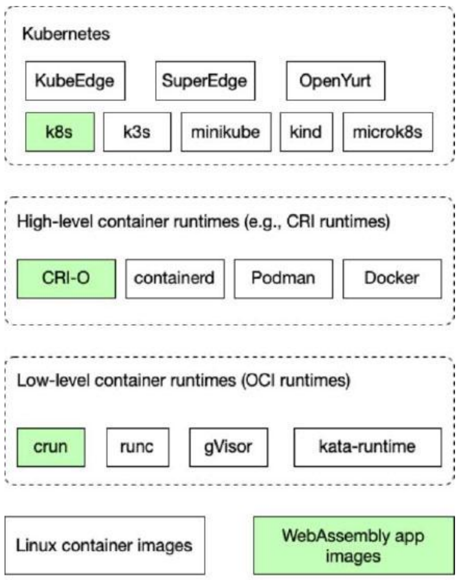
二、云原生 WebAssembly

1、与现有云原生生态的集成

crun 支持 WasmEdge 作为容器 Runtime。



在 Kubernetes 里管理 WebAssembly 和 Linux Container，下图是 Kubernetes 的生态图。

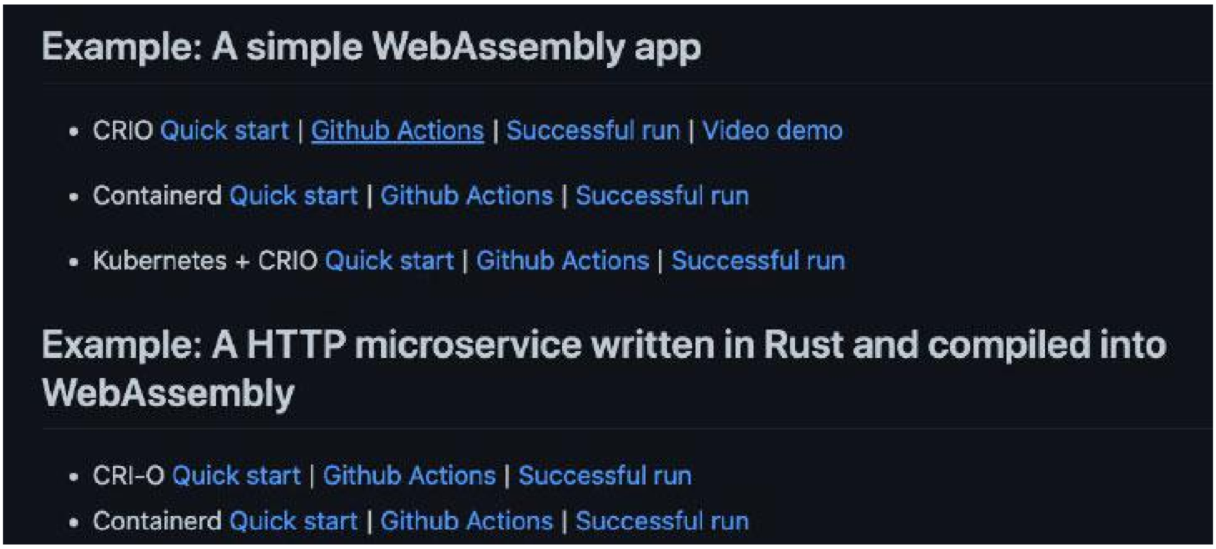


The container ecosystem

图中底层是镜像，可以是 Linux Container Images，也可以是 WebAssembly App Images；上一层是 OCI 状态，可以直接去运行 Images，这四个的功能都是一样的，可以直接运行 Linux

的镜像；再上一层就是 Kubernetes 状态，管理 CRI-O，一层层的下来，在 WasmEdge 之前，只有一种容器镜像的 container，叫做 Linux Container。加了 WebAssembly 后，我们跟 crun 的开发者合作，使得 crun 里边支持 WebAssembly ，将 Docker Images 和 WebAssembly Images 都 crun pull 下来，因为 crun 做了改动，Images 就可以放到对应的位置上去运行，一旦 crun 支持，整个生态就都可以支持了。

参考资料：WasmEdge in Kubernetes：https://wasmedge.org/book/en/kubernetes.html


Example: A simple WebAssembly app

- CRIO Quick start | Github Actions | Successful run | Video demo
- Containerd Quick start | Github Actions | Successful run
- Kubernetes + CRIO Quick start | Github Actions | Successful run

Example: A HTTP microservice written in Rust and compiled into WebAssembly

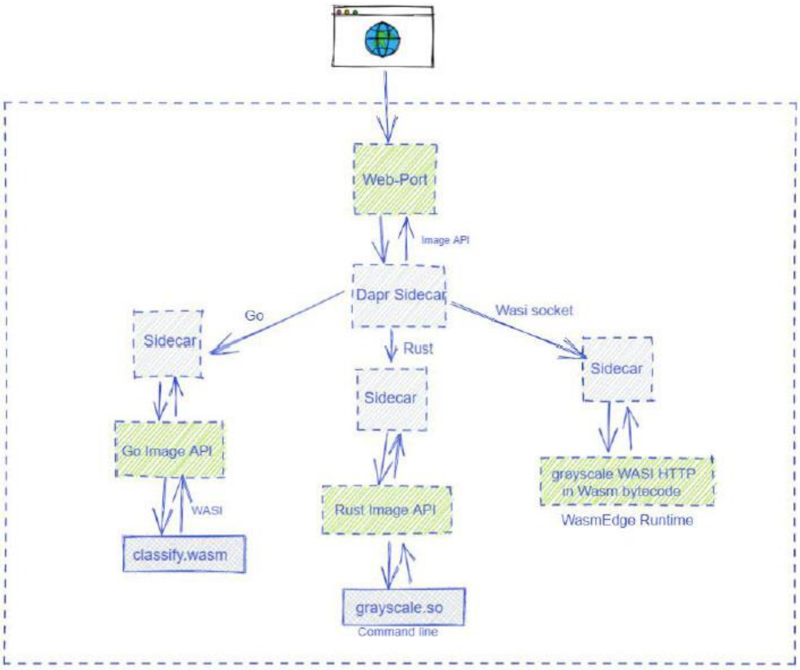
- CRI-O Quick start | Github Actions | Successful run
- Containerd Quick start | Github Actions | Successful run

2、与 Service Mesh 的集成

Sidecar 微服务

- 通过 WasmEdge 的 language SDK 将 WasmEdge 程序嵌入到 sidecar 中；
- 直接使用容器工具管理 WebAssembly 微服务；

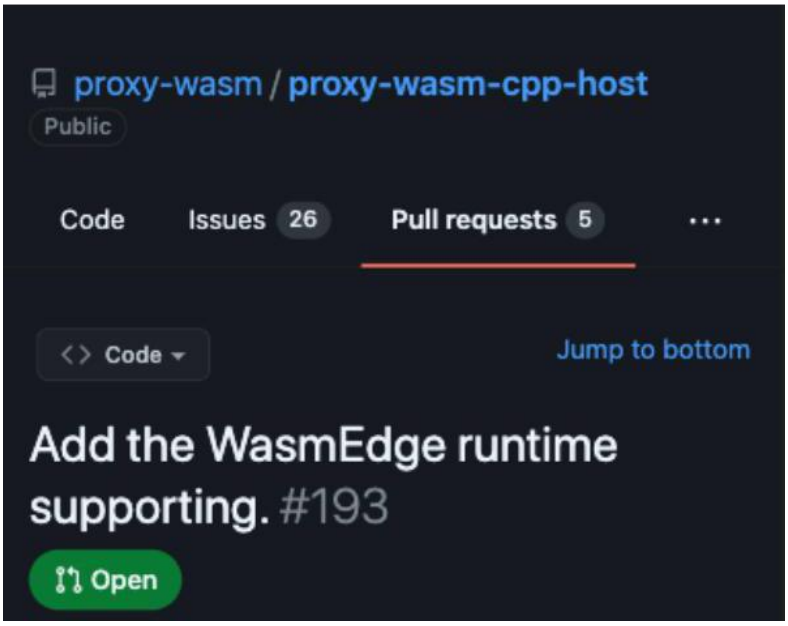
一旦和 Kubernetes 有了结合就都可以结合了，Service Mesh 里 Kubernetes 作为控制面，能够启各种各样的服务。目前是第一个也是唯一一个可以整合到里 Dapr 边去，作为 Dapr 微服务的 WebAssembly 状态。



WasmEdge + Dapr

API Proxy 的扩展

- 使用 WebAssembly 而不是 Lua 来管理和重定向 API 请求；
- 标准：Proxy wasm 规范；
- WasmEdge 支持 Proxy Wasm 规范；

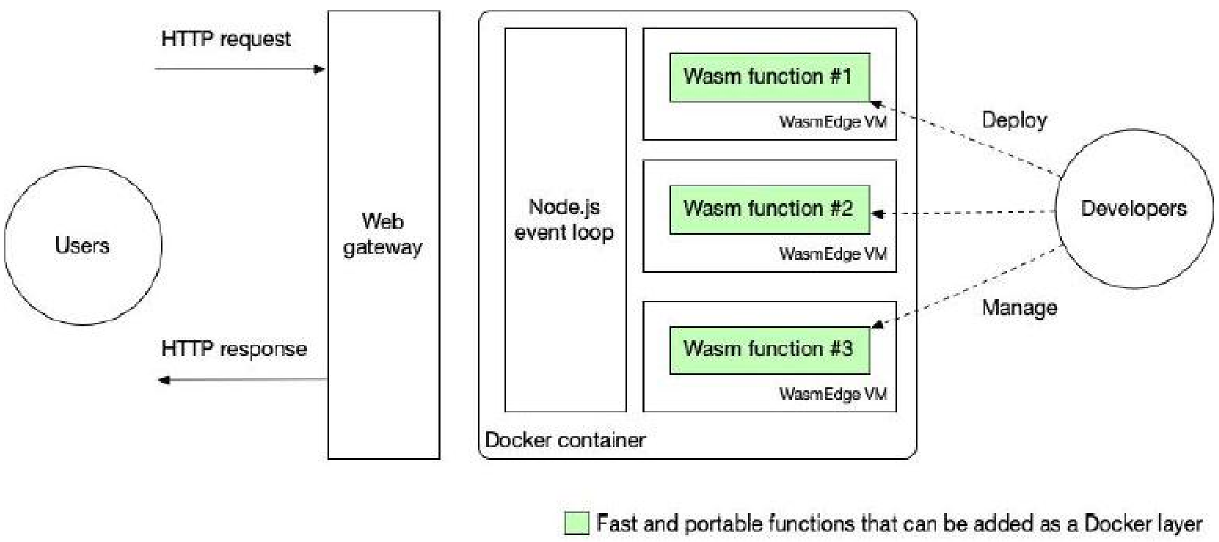


在 Service Mesh 里，在做 serverless 的时候经常会用到 API Proxy，用 WebAssembly 的话，就可以直接把用 Rust 或者 C++的程序，编译成 WebAssembly 后直接嵌入到 API Proxy 里边去。

3、与 FaaS/Serverless 的集成

公有云里的 Serverless WebAssembly

- 在 Docker 里运行 Wasm，不是最好的做法；
- 但依然有下面的好处：
  - 可移植；
  - runtime 隔离；
  - 易于部署和管理；
  - 标准的 Tensorflow Rust SDK；



在 Docker 里边做集成，很大的好处是取代了传统的非常大的 Docker images。我们未来想做的是想把 WebAssembly 和 Linux Container 可以 side by side 的运行。

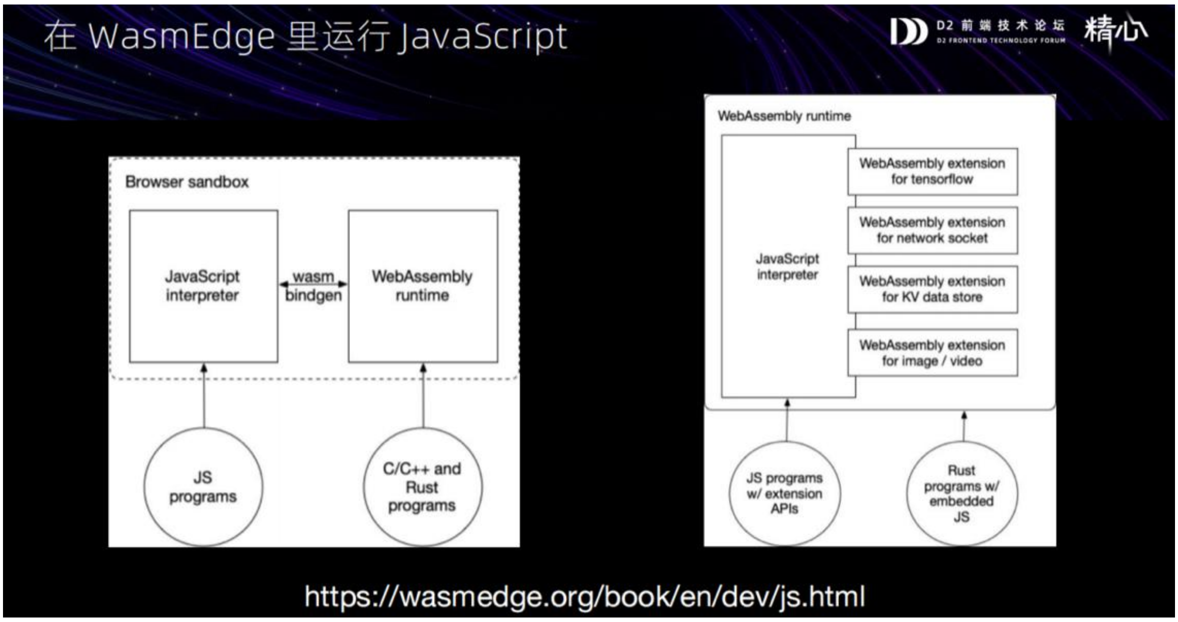
参考链接：https://wasmedge.org/book/en/frameworks/serverless.html

由 Pure WebAssembly 驱动的 Serverless

- 轻量级，没有冷启动
- 资源消耗小
- Runtime 隔离

三、WebAssembly 生态的创新

1、在 WasmEdge 里运行 JavaScript



WebAssembly 的容器和镜像，如何跟 Kubernetes、Service Mesh、Dapr、这些云原生架构去集成？之前都是将 JavaScript 放在 node.js 里边运行，现在为什么要将 JavaScript 放在 WasmEdge 里边运行？放在 WasmEdge 里边运行有什么好处呢？

V8 有三个大的问题：

- V8 复杂而大，是为浏览器写的，如果只是运行一些简单的 JavaScript，不适用；
- V8 不安全，安全性主要是都是为浏览器所设计的，在服务端存在一些安全隐患，即便在 Docker 里运行起来，运行的效率也极其低；

- V8 可调度性差；

在 WasmEdge 里边运行 JavaScript 是一个比较叛逆的工作，在这个市场没有其他人去做。我们这个是最好的：

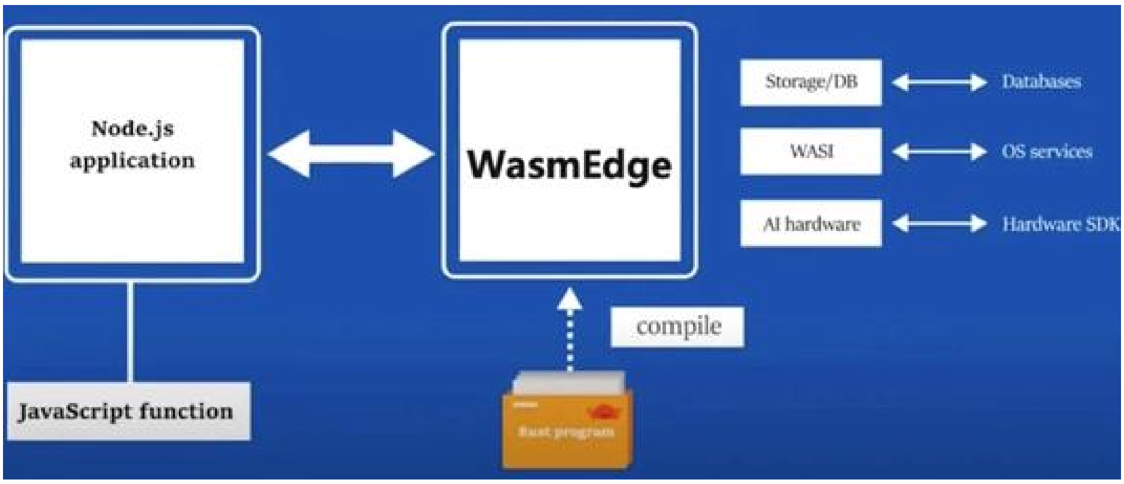
- 支持 ES6 module 和 std API；
- 用 Rust 实现 JS API；
- 将本地 C 共享库导出为 JS 函数；

无论在可管理性，在执行的效率，用 WasmEdge 运行 JavaScript 都比 V8 好很多。做边缘云开发的都对这个感兴趣很多。

通过 WebAssembly 的整合给我们带来的好处，大大增建了 WebAssembly 的性能，变轻变安全，能够直接进行安全交互，大大提高 JavaScript Application 的性能。

source code: https://github.com/second-state/wasmedge-quickjs
参考链接：https://wasmedge.org/book/en/dev/js.html

2、将 WasmEdge 嵌入 Node.js



- Host 应用程序是用 JavaScript 编写的 Node.js Web 应用程序，负责 WebAssembly 函数调用；

- WebAssembly 字节码是用 Rust 编译的，在 WasmEdge runtime 内部运行，并从 Node.js Web 应用调用；

参考链接：https://wasmedge.org/book/en/embed/node.html

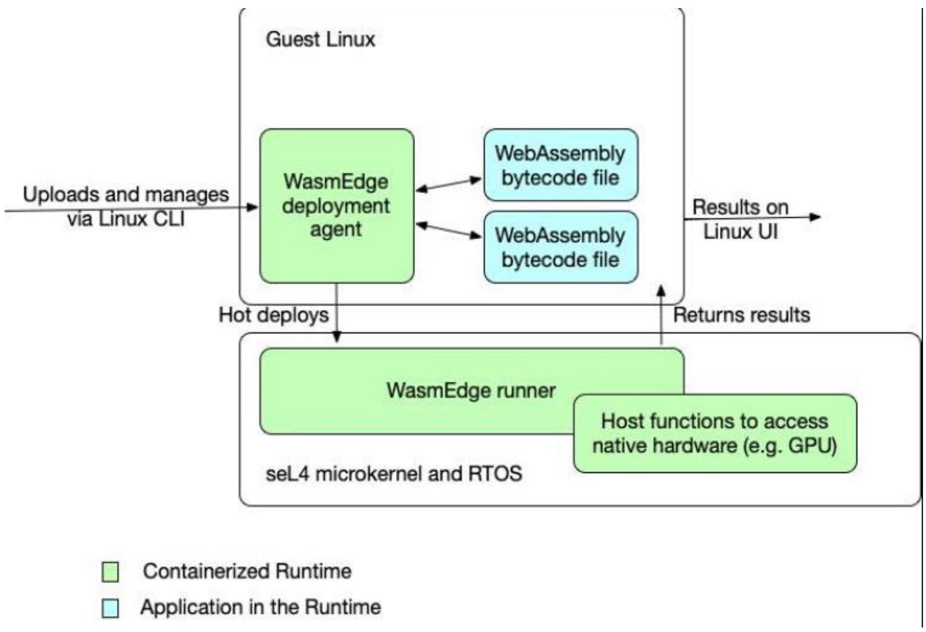
3、WASI Socket

WebAssembly 也存在一些问题，目前官方的 Socket 仍然是争论不休，但有了 wasmedge_wasi_socket，就可以用 Rust 编写 networking 应用，并编译成 Wasm 字节码。

source code: https://github.com/second-state/wasmedge_wasi_socket
Rust：https://wasmedge.org/book/en/dev/rust/networking.html
JavaScript：https://wasmedge.org/book/en/dev/js/networking.html

4、实时操作系统上的 WasmEdge

- 边缘上的容器；
- 更适用于资源受限的边缘设备；
- 通过 WebAssembly 扩大 seL4 的开发者社区；



对 WebAssembly 来讲一个很大的卖点就是跨平台性，可以在不同的操作系统和不同硬件上去运行，这也是对 WebAssembly 能力的一个认可。

https://wasmedge.org/book/en/
https://github.com/WasmEdge/WasmEdge
入群交流：CNCF S lack # WasmEdge

Node.js Web 框架再进化 — 面向前端与未来标准

摘要：Web 开发一直是 Node.js 的主流方向，无论新人必学的 Express / Koa，或者是社区流行的企业级框架 Egg / Nest，各类 Web 框架层出不穷。本次将分享 Node.js Web 框架的发展历程，分析各类框架的适用场景及利弊，并基于阿里的 Node.js 框架 Midway，为大家介绍在过去这两年，我们对下一代 Node.js Web 框架的思考、设计、实践，包含如何面向前端做一款前端“爱用”的 Node.js 框架，如何面向未来标准甚至参与标准来设计 Node.js Web 框架两部分。

分享人：刘子健)繁易)，阿里巴巴前端技术专家，Node.js 框架 Midway 负责人，Node.js Core Collaborator，阿里巴巴 TC 39 代表，专注于 Node.js 架构 / Serverless / 开发者体验 / 前端标准 / 工程化领域，具有丰富的实战经验与优秀的技术视野。

目录：

一、Node.js & Web 框架简述
二、Midway - 面向前端的框架演进之路
三、未来-面向标准&规划

一、Node.js & Web 框架简述

Node.js 是基于 Chrome V8 JavaScript 引擎的运行时，一般用于编写 CLI、处理数据、编写 Restful API、进行页面渲染等等多种作用。之前 JavaScript 在浏览器端限制比较多，但是自从有了 Node.js 以后，应用范围得到很大拓展。

1、Web 框架功能

现代的 Web 开发不管使用什么语言都离不开 Web 框架。Web 框架具有 Restful API、数据库 CRUD、页面渲染、身份校验等功能，提供了高效开发 Web 应用的方式，同时 Web 框架存在适

用场景以及规则约束，有源源不断的框架推陈出新。

2、Node.js 框架的发展阶段

Node.js 的发展分成三个阶段，分别是起步期、企业架构期、面向前端期。

起步阶段，主打轻量

2009 年 Node.js 出现，2010 年 Express 框架出现，2013 年出现 Koa 框架。当时前端工程师主要是在尝鲜，不敢在业务上做太多的尝试和落地，更多的是验证 Node.js Web 场景可行性。在起步期阶段，框架主打轻量和极简。下面是两个框架的写法示例。

```
const app = require('express')()

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(3000)
```

```
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

左侧是 Express 框架，右侧是 koa 框架

其优点是简单易学，易于集成，Express 框架容易集成到 Nest 和 Webpack 框架中，Koa.js 框架容易集成到 Egg 和 Midway 框架中，生态繁荣，久经考验。

其缺陷也是比较明显，表现为缺乏规范和最佳实践、不利于团队协作和大规模开发、Express 年久失修。

企业阶段，主打架构

在 2014 年到 2017 年，Node.js 规模化落地，专业的 Node.js 工程师出现，主打企业级框架和架构以及规模化和团队协作化。

在这一段时间主要出现的框架有 nest、Egg、Midway 框架，但大多以 Express 和 Koa 框架作为基础框架，如下图：



企业级框架优点为大而全，功能完善，规范并且最佳实践明确，易于团队协作，同时社区生态活跃。

缺陷是大而全导致上手成本高；限制多，难扩展。

面向前端

从 2016 之后，Node.js 发展成熟完善，Node 4.0 发布，前端工程师人数急速增加，主打面向前端框架的设计，以及简洁和轻量，在这个阶段中主要框架是 Next.js 以及 Nuxt.js 框架。

这些框架的优点主要是来自于前端，全栈开发，简单容易学习，支持 Serverless 部署。
其缺陷是后端功能较弱，自定义扩展困难，强依赖于平台支持。

下面是两个 Demo 示例：

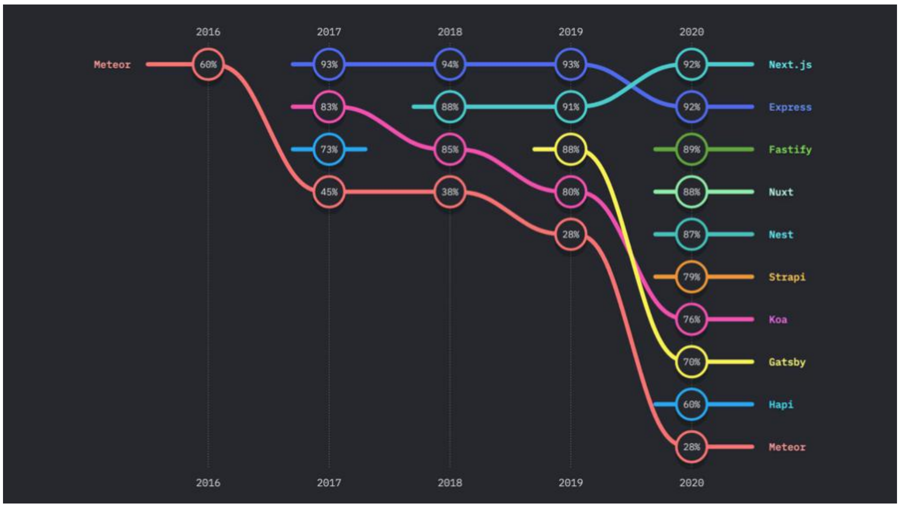
```javascript
export default (req, res) => {
  res
    .status(200)
    .json({ name: 'John Doe' })
}
```

```javascript
export default async (req, res) => {
  await someAsyncFunction()

  return {
    someData: true
  }
}
```

左侧是 Next.js，右侧是 Nuxt.js

在 2020 年的时候，一份 stateofjs 2020 关于 Node.js Web 框架满意度对比的调查报告表明，Next.js 成功登顶，并且 Express.js 框架仍然受到关注。



Node.js Web 框架满意度调研（stateofjs 2020）

3、总结

- Node.js Web 框架迭代与前端行业发展密切相关；
- 前端应用场景多于纯 Node.js 后端场景；
- 面向前端设计的全栈框架兴起，使得 Node.js 用法回归简洁和轻量。

二、Midway - 面向前端的框架演进之路

Midway 是 2014 年开发的框架，发布了 7 个大版本，2018 年正式开源。

Midway 作为一款企业级开发框架，技术选型上，主要有 TypeScript（静态类型、多人协作）、IoC（复杂架构、面向接口编程）、Egg（统一框架、复用生态）。Midway Demo 示例如下图：

```
import {
  Inject,
  Controller,
  Get,
  Provide,
  Query,
} from '@midwayjs/decorator';
import { UserService } from '../service/user';

@Provide()
@Controller('/api/user')
export class APIController {
  @Inject()
  userService: UserService;

  @Get('/')
  async getUser(@Query('id') uid) {
    const user = await this.userService.getUser(
      uid
    );
    return {
      success: true,
      message: 'OK',
      data: user,
    };
  }
}
```

1、Node.js 应用状况

集团在 2019 年的时候，Node.js 应用状况表现为：

- **服务器利用率低：**集团 1600+Node.js 应用，常年 cpu 利用率<10%，乃至 5%，服务器利用率低。
- **DevOps 成本高：**前端维护乏力，由于是 Docker 应用、限流、日志、跨语言，所以导致 Devops 成本过高。

传统应用的缺点限制了 Node.js 在阿里的进一步发展。

2、前端诉求

- 后端往大后台下沉，前端往小前台发力，提升生产力。
- 前端同学希望将中台服务快速组合为各类业务接口，和端侧同步快速交付前台，以更快的响应业务需求变化来帮助业务试错。

总体上需要赋能前端，让云原生给前端降本增效。

3、技术方向

通过以上现状和诉求，2019-2020 阿里经济体前端委员会的四大技术方向为：

- 搭建服务；
- Serverless；
- 智能化；
- IDE。

4、面临挑战

- 用户群体割裂，如何在一个框架下服务好两种用户？

前端工程师和 Node.js 工程师出现割裂。前端工程师偶尔写接口，希望简单易上手；对于 Node.js 工程师来说，因为日常管理上万行代码，更注重于复杂场景的应对能力。

- 使用场景不同，如何在一个框架下支持两种场景？

简单场景和企业级场景不同，简单场景要求快速实现 CRUD，接口聚合；企业级场景注重维护性，依赖注入，基础架构，简单场景将会逐步演化成为复杂场景。

- 前端范式变更

目前主流的前端都在从 Class Component 向 Function + Hooks 转变，包括 React、Vue 等，最终会导致函数式开发和 OOP 开发并存，如下图比较图：

```jsx
import { useState } from 'react';

function Counter() {
  const [value, setValue] = useState(0);

  function onButtonClick() {
    setValue(counterValue + 1);
  }

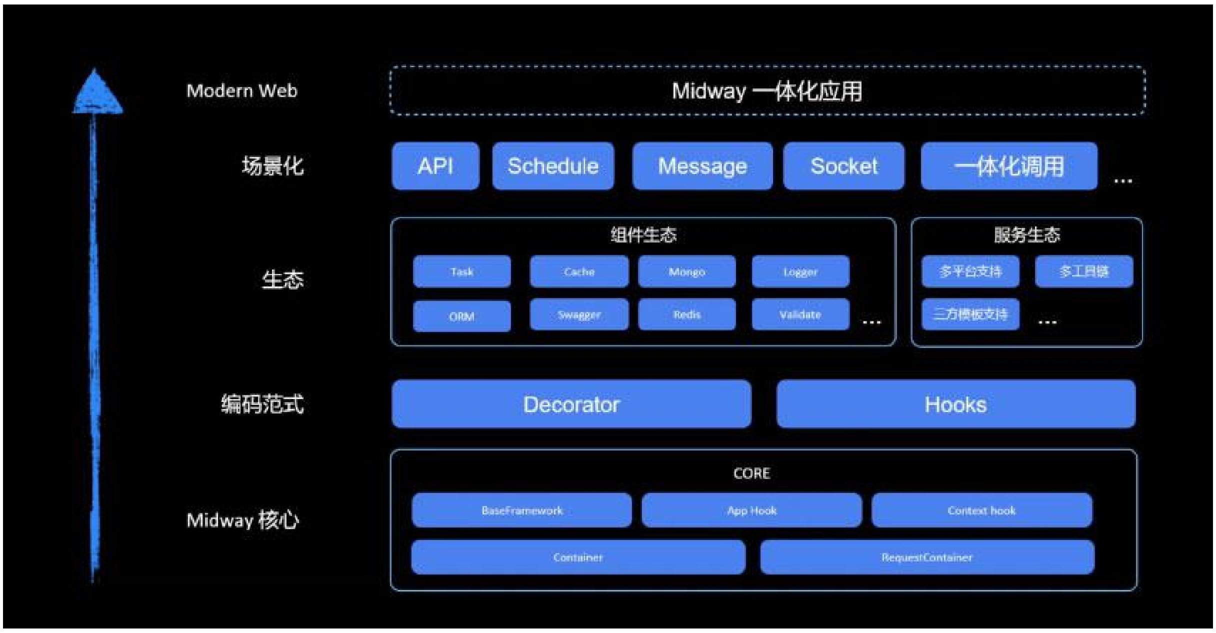
  return (
    <>
      <div>{value}</div>
      <button onClick={onButtonClick}>+1</button>
    </>
  );
}
```

```ts
import {
  Controller,
  Get,
  Provide
} from '@midwayjs/decorator';

@Provide()
@Controller('/')
export class HomeController {
  @Get('/')
  async home() {
    return 'Hello Midwayjs!';
  }
}
```

同一个开发者在写前、后端的时候，思维是不一样的，那么框架有没有可能支持函数式开发，又能与 OOP 并存?

5、解决办法：渐进式设计

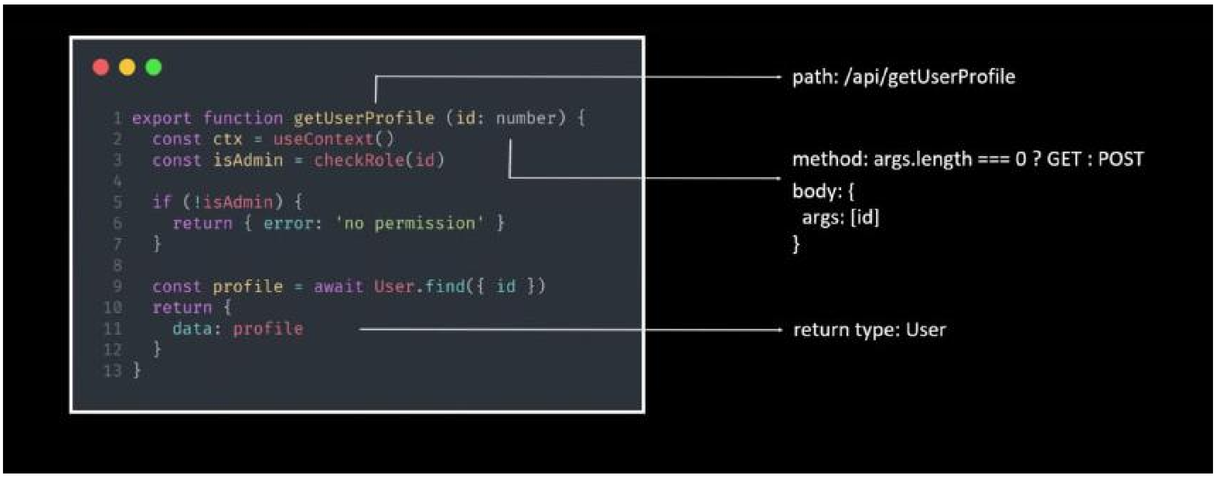


通过 Midway 的渐进式设计来解决，我们把 Midway 架构进行分层，从下到上主要为 Midway 核心、编码范式、生态、场景化、和 Modern Web。对于前端来说的解决方案就是 Hooks。

Hooks

- **函数即接口**

JavaScript 函数即接口，统一并且无协议。



函数名称生成路径，可以根据参数的长度决定请求类型，通过 TyperScript 可以推断返回类型。通过这种方式，基于函数元信息生成接口。

- **获取请求上下文**

针对前后端范式不一致的问题，设置了 useContext API，可以直接拿到运行时的参数，最终实现无需手动传入参数，就能够实现前端保持一致。示例代码如下：

```
import { useContext } from '@midwayjs/hooks'

export async function getQuery () {
  const ctx = useContext()
  return ctx.query
}
```

获取 URL 查询参数

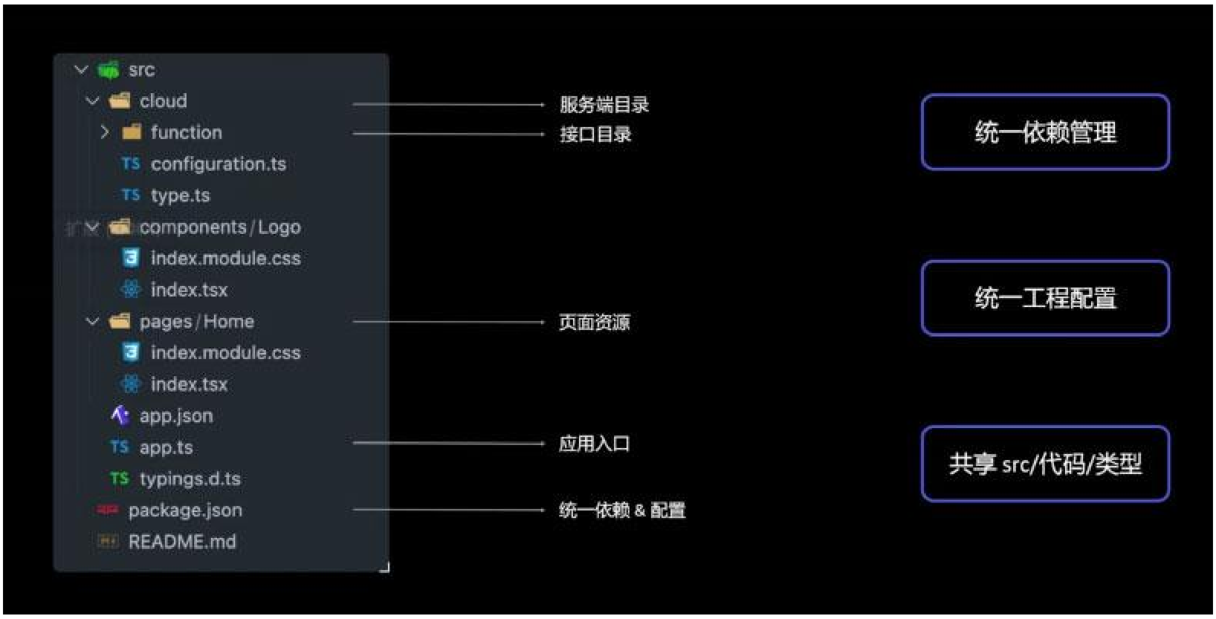
```
import { useContext } from '@midwayjs/hooks'
import { useRequest } from './useRequest'

export async function demo() {
  const ctx = useContext()
  const request = useRequest()

  return {
    url: ctx.url,
    header: request.headers
  }
}
```

自定义 Hooks

- **面向全栈应用设计**



典型工程代码目录

上图是典型工程代码目录，包括服务端目录、接口目录、页面资源、应用入口等，都在 src 下面，最终实现统一管理和配置，共享 src 代码、类型。

- **简化接口调用**




```
import { getUserProfile } from './api'
import { useAsyncEffect } from './utils'
import { useState } from 'react'

export default () => {
  const [name, setName] = useState('')
  useAsyncEffect(async () => {
    const profile = await getUserProfile()
    if (profile.error) {
      alert(profile.error);
    }
    setName(profile.name)
  }, [])

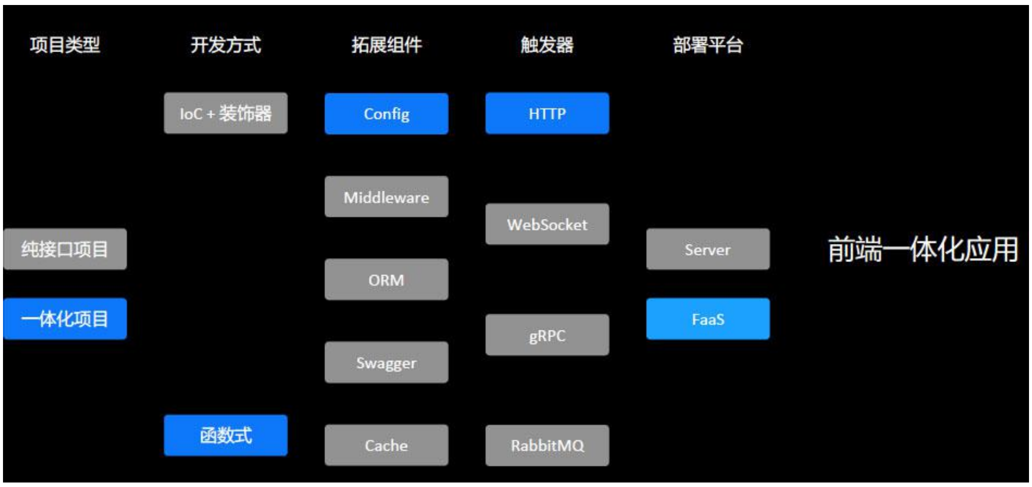
  return <div>{name}</div>
}
```

“零” API 调用

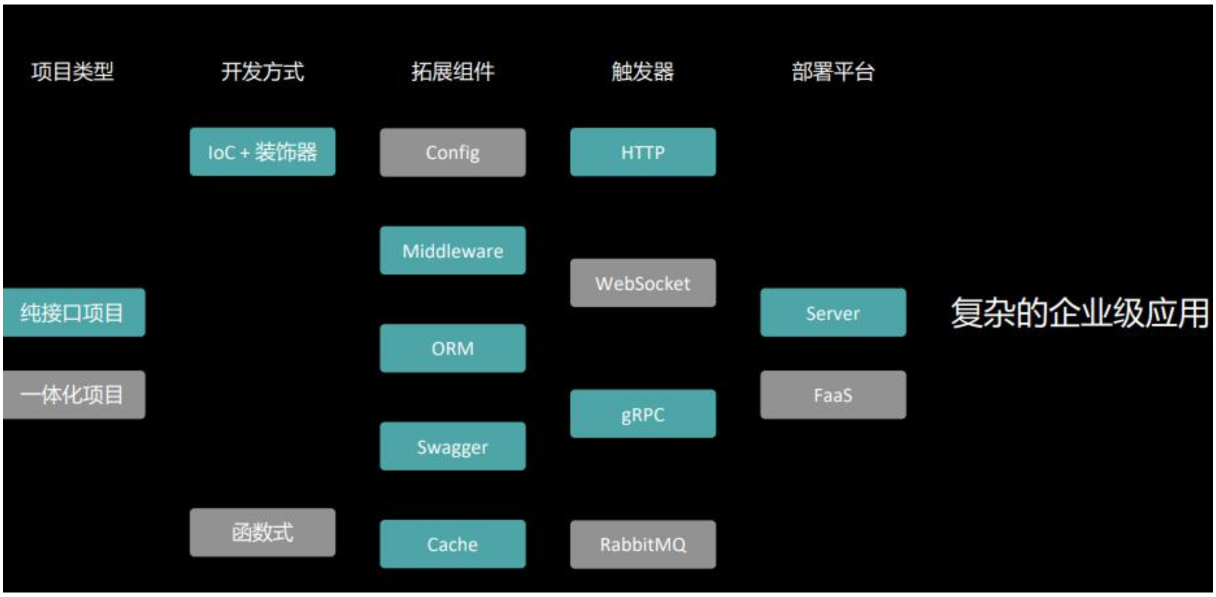
和传统接口调用不同，在一体化应用中，只需要导入函数，调用函数并使用返回值，因为都在一个 src 目录，且都是 TypeScript，这种直接导入的方式是可行的，同时后端的返回值也不需要生成或推断，最终实现“零”API 调用，节约时间，节省文档，实现直接调用。

渐进式-类似于搭积木一样进行演化

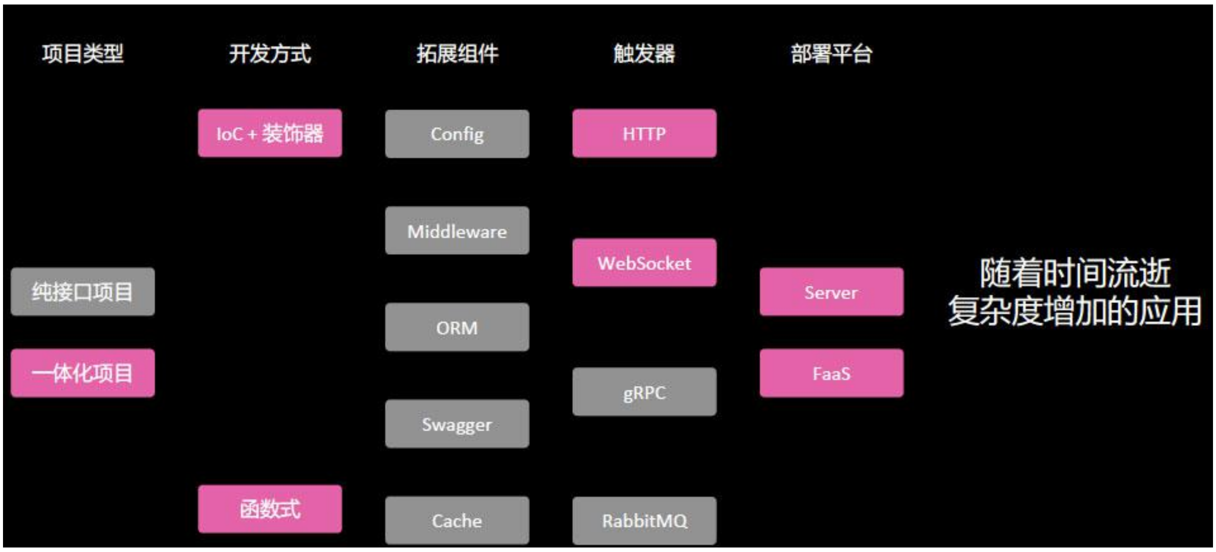
把 Midway 支持的功能按照项目类型、开发方式、拓展组件、触发器、部署平台进行罗列，可以像像搭积木一样，选择相应的积木就可以实现。比如面向前端一体化应用，可以选择一体化项目、函数式、Config、HTTP、FaaS，如下图：



如果是面向后端，可以使用 IoC+装饰器、Middleware、ORM、Swagger、Cache、HTTP、gRPC，部署在 Server 上，实现复杂的企业级应用。



也可以面向随着时间流逝复杂度增加的应用，如下图：



6、落地情况



Hooks 于 2020 年 4 月发布，有 2500 多个应用，是阿里前端的主流模式。

7、总结

- 企业内仍存在简单场景与复杂场景，框架设计应考虑到此问题；
- Node.js Web 框架应关注开发者体验，面向前端工程师设计；
- 云 + 端的研发模式将成为未来的主流研发模式。

三、未来-面向标准&规划

1、类型安全

例如社区的 Prisma 框架，可以生成 ORM，ORM 可以根据数据库实时生成。如果前端也是自动生成，那么将会实现前端到后端再到数据库全链路的类型安全，如果后端修改了，那么前端也将会自动报错。

2、展望：云端融合

目前有两个在推的提案，分别为 JS Module Blocks（动态引入代码）和 JS Module Fragments（模块可以命名，模块可以静态导入）。



根据这两个提案，引入我们的第三个方向，最终实现在一个文件中，可以实现 server 端和 ssr 端以及 client 端融合。

```
export module server {
  import { prisma } from './prisma'

  // get articles from database
  export async function getArticles (type: string) {
    const articles = await prisma.articles.find({
      where: { type }
    })
    return articles
  }
}

export module ssr {
  import { useContext } from '@midwayjs/hooks'

  // fetch github stars for server side rendering
  export async function getServerSideProps () {
    const ctx = useContext()
    const stars = await fetchGithubStars(ctx.query.name)
    return props: { stars }
  }
}

export module client {
  import { useFetch, PromiseType } from './utils'
  import { getArticles } from server
  import { getServerSideProps } from ssr

  type Props = PromiseType<typeof getServerSideProps>

  export default function App (props: Props) {
    const { data } = useFetch(getArticles)
    return <h1>Stars: {props.stars} Articles: {data}</h1>
  }
}
```

方向示例

目前正在与前端委员会标准化小组推进 TC39 提案，反馈场景，谋求推进至 Stage 2。

项目网址：https://github.com/tc39/proposal-module-fragments/issues/14

用 Rust 和 Node-API 开发高性能 Node.js 模块

摘要：随着 Deno 项目横空出世，Rust 逐渐被带入到 Node.js 开发者的视野中。而 Deno 又推动了 SWC 的迅速发展，使其在多个重量级项目中取代了 Babel 成为了默认的编译工具。要想用 Rust 为前端与 Node.js 领域带来无与伦比的性能体验，如何让 Rust 代码与 Node.js 进行交互是绕不开的话题。本次分享将会为大家介绍如何从零开始编写、构建、发布一个完整的由 Rust 编写的 npm 包，并为大家详细讲解每一个环节的要点和细节。

分享人：龙逸楠，字节跳动前端工程师，TypeScript 与 Rust 爱好者，先后就职于 Teambition，ARKIE，LeetCode。热爱开源，近期在探索如何通过 Rust 提升前端与 Node 的开发体验与性能。

一、Node.js 中的 Native Addon

1、Native Addon 的本质

- .node 格式的文件；
- 是二进制文件，无法用文本编辑器直接阅读；
- 是一个动态链接库(Windows DLL/Unix dylib/Linux so)；

参考链接：https://xcoder.in/2017/07/01/nodejs-addon-history/

2、.node 格式文件

Node.js 对 .node 格式文件的加载方式:

modules/cjs/loader.js

```
// Native extension for .node
Module._extensions['.node'] = function(module, filename) {
  if (policy?.manifest) {
    const content = fs.readFileSync(filename);
    const moduleURL = pathToFileURL(filename);
    policy.manifest.assertIntegrity(moduleURL, content);
  }
  // Be aware this doesn't use `content`
  return process.dlopen(module, path.toNamespacedPath(filename));
};
```

上图橙色方框中代码：

- Node.js 默认将 Native Addon 当作 CommonJS 格式模块加载；
- 使用 process.dlopen 函数加载内容；
- 目前源码包含对 ESM 的 import assets 逻辑的兼容，但实际上 Node.js 目前的最新版（17.1.0)只支持 json 类型的 import assert；

参考网址：
https://github.com/nodejs/node/blob/2cc7a91a5d855b4ff78f21f1bb8d4e55131d0615/lib/internal/modules/cjs/loader.js#L1172

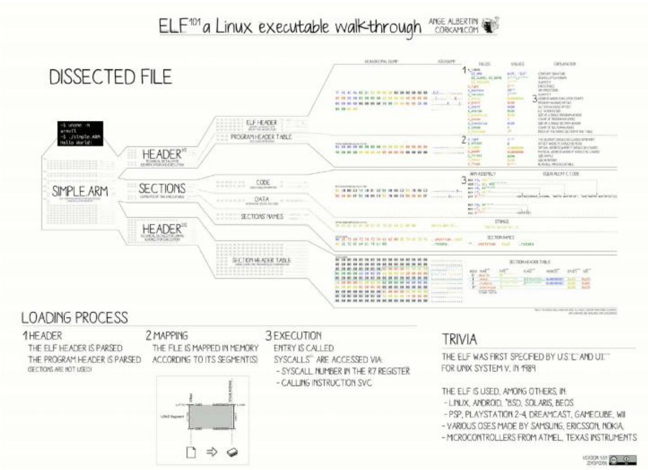
3、Native Addon 文件内容

在 Linux 上使用 hexyl 查看 blake.darwin-x64.node 文件内容。File ELF 文件指向 0×03，e_type，ET_DYN 类型文件（图中橙色框），也就是动态链接库。

```
hexyl -n 256 node_modules/@napi-rs/blake-hash-linux-x64-gnu/blake.linux-x64-gnu.node

00000000 | 7f 45 4c 46 02 01 01 00 | 00 00 00 00 00 00 00 00 | •ELF•••0 | 00000000
00000010 | 03 00 3e 00 01 00 00 00 | 80 e5 01 00 00 00 00 00 | •0>0•000 | ××•00000
00000020 | 40 00 00 00 00 00 00 00 | b0 59 0a 00 00 00 00 00 | @0000000 | ×Y_00000
00000030 | 00 00 00 00 40 00 38 00 | 0b 00 40 00 20 00 1f 00 | 0000@080 | •0@0 0•0
00000040 | 06 00 00 00 04 00 00 00 | 40 00 00 00 00 00 00 00 | •000•000 | @0000000
00000050 | 40 00 00 00 00 00 00 00 | 40 00 00 00 00 00 00 00 | @0000000 | @0000000
00000060 | 68 02 00 00 00 00 00 00 | 68 02 00 00 00 00 00 00 | h•000000 | h•000000
00000070 | 08 00 00 00 00 00 00 00 | 01 00 00 00 04 00 00 00 | •0000000 | •000•000
00000080 | 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00 | 00000000 | 00000000
00000090 | 00 00 00 00 00 00 00 00 | 44 d5 01 00 00 00 00 00 | 00000000 | D×•00000
000000a0 | 44 d5 01 00 00 00 00 00 | 00 10 00 00 00 00 00 00 | D×•00000 | 0•000000
000000b0 | 01 00 00 00 05 00 00 00 | 44 d5 01 00 00 00 00 00 | •000•000 | D×•00000
000000c0 | 44 e5 01 00 00 00 00 00 | 44 e5 01 00 00 00 00 00 | D×•00000 | D×•00000
000000d0 | ec 2f 08 00 00 00 00 00 | ec 2f 08 00 00 00 00 00 | ×/•00000 | ×/•00000
000000e0 | 00 10 00 00 00 00 00 00 | 01 00 00 00 06 00 00 00 | 0•000000 | •000•000
000000f0 | 30 05 0a 00 00 00 00 00 | 30 25 0a 00 00 00 00 00 | 0•_00000 | 0%_00000
```

参考网址：https://en.wikipedia.org/wiki/Executable_and_Linkable_Format



ELF 的扩展链接：

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format#/media/File:ELF_Executable_and_Linkable_Format_diagram_by_Ange_Al bertini.png

Node.js 原生扩展加载后的执行主要通过 napi_register_module_v1 函数，具体处理过程如下：

- process.dlopen；
- 初始化 env 与 module.exports 对象，并传入 napi_register_module_v1 函数；
- 拿到返回的 module.exports 对象，后续将 native addon 当作普通的 CommonJS 模块处理；

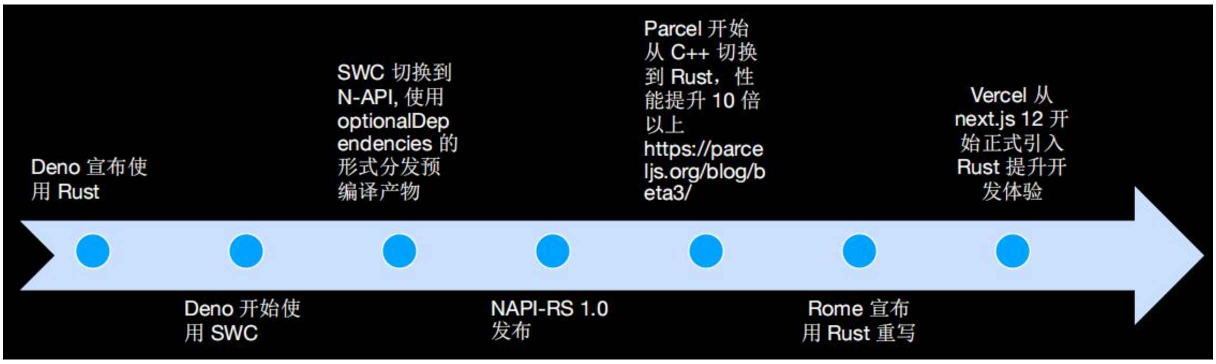
```
extern "C" {
  napi_value napi_register_module_v1(napi_env env, napi_value export);
}
```

参考链接：https://github.com/nodejs/node/blob/v10.23.0/src/node.cc#L1232

二、Rust 在前端/Node.js

1、Rust 在前端/Node.js 领域应用的历史

- Rust 语言进入前端开发者视线是从 2018 年 Deno 宣布使用 Rust 开始；
- 接着 Deno 使用 SWC 做编译工作；
- SWC 切换到 N-API，使用 optionalDependencies 的形式分发预编译产物是一个关键的节点，在这之后 SWC 开始在 Node.js 领域被广泛使用；
- NAPI-RS1.0 发布之后前端和 Rust 很多工具都使用 NAPI-RS 作为桥梁将 Rust 语言引入；
- Parcel2.0 发布，开始从 C++切换到 Rust，使性能提升 10 倍以上（参考网址：https://parceljs.org/blog/beta3）；
- Rome 宣布用 Rust 重写；最近 Vercel 从 next.js 12 开始正式引入 Rust，使开发体验得到质的提升。



参考网址：https://github.com/denoland/deno/issues/208#issuecomment-395739171

2、目前流行的 Rust 编写的前端/Node.js 库

- Parcel 使用 SWC 与 NAPI-RS 实现 JavaScript/TypeScript 编译与优化逻辑，如 Tree-shaking 与 Scope hoist，minify 等，source map 模块、png/jpeg 无损压缩、css 压缩与编译模块，也是由 Rust + NAPI-RS 编写；
- Rome 是一个集 Linter/Compiler/Bundler/Formatter/Test Runner 为一体的工具链，目的是取代 Babel、ESLint、Webpack、Jest、Prettier 等前端工具链，最初由 TypeScript 编写，现已全面切换到 Rust；
- Next.js 是 Vercel 出品的、前后端一体化 React 框架，目前它将所有定制的 Babel custom transformer 与默认的 TypeScript/JavaScript 编译部分切换到 Rust，使开发体验得到大幅提升；
- Prisma 是下一代面向 TypeScript/JavaScript 的 ORM，Prisma 的数据库连接与查询引擎使用 Rust 与 NAPI-RS 实现，性能提升 20+倍；
- Ali ICE，Shopify FE 等团队使用 SWC 替换自定义 Babel plugin。

3、如何发布 Rust 编写的前端/Node.js 工具链

目前流行的 Native Addon 发布方式有以下三种：

- 只发布源码，通过 postinstall 脚本执行编译脚本，在安装的时候编译成二进制；
- 在 CI 阶段预编译不同平台的二进制文件，在 postinstall 的时候下载下来；
- 在 CI 阶段预编译不同平台的二进制文件，将不同平台的二进制文件单独发布成包，然后通过 package.json 中的 CPU 与 OS 字段限制这些包可安装的平台，最终将所有平台的独立包指定为最终发布包的 optionalDependencies，主流的包管理工具比如 npm/pnpm/yarn 就可以通过预设的 CPU 与 OS 字段只下载对应平台的二进制包。

4、只发布源码

优点

- 对于库开发者来说，无需任何额外的工具链维护成本，只需要改动源码就可以轻松发布；

- 在用户的机器上用用户的工具链编译，兼容性是最好的。只要编译通过，基本上就可以运行，不会有系统组件比如 GLIBC libstdc++ 等不兼容的情况；

缺点

- 大型项目的编译时间过长，并且占用大量 CPU 资源；
- 可能需要用户安装编译相关的工具链，比如 GCC CMAKE LLVM NASM Rust 等，前端开发者一般缺乏相关经验，对于很多项目很难设置正确；
- 编译期间可能产生大量中间文件，不正确清理极其占用硬盘空间；
- 让很多 CI 的缓存逻辑无法工作，不得不重新全量下载 & 编译；
- postinstall 脚本不安全，未来可能会被默认禁用（社区参考：https://github.com/npm/rfcs/pull/488）

5、发布预编译产物，postinstall 的时候下载

优点

- 对于用户来说，减少了编译成本；
- 不需要考虑编译时工具链等问题；
- 没有中间编译产物，节省磁盘空间；

缺点

- postinstall 安装产物不受 lockfile 控制，会破坏很多 CI 的缓存逻辑，降低缓存复用率；
- 存放预编译产物的 CDN 无法照顾到全球的用户，比如大量项目使用 GitHub Releases 作为预编译二进制存放的服务，在国内访问困难；
- 因为要安装时下载，需要引入运行时不需要的依赖来完成下载，比如 node-pre-gyp 等；
- postinstall 脚本不安全，未来可能会被默认禁用（社区参考：https://github.com/npm/rfcs/pull/488）

6、optionalDependencies

下面代码图中左边是要发布的包“black hash”，在 optionalDependencies 上声明了很多包，除了 darwin-arm64 其它包都不符合 CPU 和 OS 限制，则只有 OS 是 darwin，CPU 是 arm64 的包会被下载（见右图），这就是 optionalDependencies 分发原理。

```
{
  "name": "@napi-rs/blake-hash",
  "version": "1.3.0",
  "main": "index.js",
  "optionalDependencies": {
    "@napi-rs/blake-hash-win32-x64-msvc": "1.3.0",
    "@napi-rs/blake-hash-darwin-x64": "1.3.0",
    "@napi-rs/blake-hash-linux-x64-gnu": "1.3.0",
    "@napi-rs/blake-hash-darwin-arm64": "1.3.0",
    "@napi-rs/blake-hash-android-arm64": "1.3.0",
    "@napi-rs/blake-hash-linux-arm64-gnu": "1.3.0",
    "@napi-rs/blake-hash-linux-arm64-musl": "1.3.0",
    "@napi-rs/blake-hash-win32-arm64-msvc": "1.3.0",
    "@napi-rs/blake-hash-linux-arm-gnueabihf": "1.3.0",
    "@napi-rs/blake-hash-linux-x64-musl": "1.3.0",
    "@napi-rs/blake-hash-freebsd-x64": "1.3.0",
    "@napi-rs/blake-hash-win32-ia32-msvc": "1.3.0"
  }
}
```

```
{
  "name": "@napi-rs/blake-hash-darwin-arm64",
  "version": "1.3.0",
  "os": [
    "darwin"
  ],
  "cpu": [
    "arm64"
  ],
  "main": "blake.darwin-arm64.node",
  "files": [
    "blake.darwin-arm64.node"
  ],
  "license": "MIT",
  "engines": {
    "node": ">= 10"
  }
}
```

优点

- 无感知，与安装普通 JavaScript 编写的 npm 包体验上没有区别；
- lockfile 与 cache 友好；
- 没有 postinstall 脚本；

缺点

- CI 配置复杂，发布流程繁琐；
- CI 环境链接的系统组件与用户使用环境可能存在不兼容情况，比如 GLIBC 版本（社区参考：https://github.com/swc-project/swc/issues/1410）

7、使用 Rust 开发 npm 包，如何选择发布模式

推荐使用 optionalDependencies，其原因在于：

- 如果使用 Rust，其工具链体积巨大，编译缓慢，中间产物庞大，不适合在用户侧编译；
- postinstall 存在潜在的安全风险，前途未卜；
- 在 postinstall 下载预编译产物对 lockfile 与构建缓存非常不友好；
- 大量 Node 应用在企业私有网络内构建，无法访问外部 CDN，postinstall 下载失败率非常高；
- CI 的复杂配置与相关工具链的维护成本被 NAPI-RS cover 了，作为包作者无需关心。

主流包含 Native 工具链的包比如 esbuild SWC 和 next.js 都采用了 optioanlDependencies 的形式发布。

参考网址：https://github.com/evanw/esbuild/pull/1621

三、NAPI-RS 介绍

NAPI-RS 是使用 Rust 编写的 Node.js 原生扩展的框架。

官网：https://napi.rs

1、NAPI-RS 特性

NAPI-RS 的核心特性：

- 简单易用的 API；
- TypeScript .d.ts 文件生成；
- 深度集成 Rust Tokio 异步运行时，轻松复用 Rust 异步代码；
- 开箱即用的 cli，开发者无需为跨平台编译、自动化 CI 烦恼。

简单易用的 API

NAPI-RS 可以屏蔽 Node API 的调用，只需写 Rust 函数后使用 NAPI 修饰，便自动生成 JS 函数，并生成.d.ts 文件。



源码网址：https://github.com/napi-rs/node-rs/tree/main/packages/crc32

TypeScript 类型生成

- 支持 Rust 的 struct + impl 生成 TypeScript Class 对应文件；
- 联合类型与可选类型；
- 异步类型生成，Rust 的 async fn 到 TypeScript 的()=>Promise<T>；
- Rust 代码注释透传到.d.ts 文件；
- 为无法自动精确生成的 Rust 函数重写 TypeScript 类型。

参考网址：https://github.com/napi-rs/napi-rs/blob/main/examples/napi/index.d.ts

示例：TypeScript 类型生成

```rust
/// default enum values are continuos i32s start from 0
#[napi]
pub enum Kind {
  /// Barks
  Dog,
  /// Kills birds
  Cat,
  /// Tasty
  Duck,
}

#[napi]
pub struct Animal {
  #[napi(readonly)]
  /// Kind of animal
  pub kind: Kind,
  name: String,
}

#[napi]
impl Animal {
  /// This is the constructor
  #[napi(constructor)]
  pub fn new(kind: Kind, name: String) -> Self {
    Animal { kind, name }
  }
```

```rust
  /// This is a factory method
  #[napi(factory)]
  pub fn with_kind(kind: Kind) -> Self {
    Animal {
      kind,
      name: "Default".to_owned(),
    }
  }

  #[napi(getter)]
  pub fn get_name(&self) -> &str {
    self.name.as_str()
  }

  #[napi(setter)]
  pub fn set_name(&mut self, name: String) {
    self.name = name;
  }
```

```rust
  /// This is a
  /// multi-line comment
  /// with an emoji 🚀
  #[napi]
  pub fn whoami(&self) -> String {
    match self.kind {
      Kind::Dog => {
        format!("Dog: {}", self.name)
      }
      Kind::Cat => format!("Cat: {}", self.name),
      Kind::Duck => format!("Duck: {}", self.name),
    }
  }

  #[napi]
  /// This is static...
  pub fn get_dog_kind() -> Kind {
    Kind::Dog
  }
}
```

```typescript
/**
 * `constructor` option for `struct` requires all fields to be public,
 * otherwise tag impl fn as constructor
 * #[napi(constructor)]
 */
export class Animal {
  /** Kind of animal */
  readonly kind: Kind
  /** This is the constructor */
  constructor(kind: Kind, name: string)
  /** This is a factory method */
  static withKind(kind: Kind): Animal
  get name(): string
  set name(name: string)
  /**
   * This is a
   * multi-line comment
   * with an emoji 🚀
   */
  whoami(): string
  /** This is static... */
  static getDogKind(): Kind
}
```

```typescript
/** default enum values are continuos i32s start from 0 */
export enum Kind {
  /** Barks */
  Dog = 0,
  /** Kills birds */
  Cat = 1,
  /** Tasty */
  Duck = 2
}
```

示例：异步类型生成

NAPI-RS 可以定义被 Abort 的异步任务，这是其它类似框架所没有的能力。

```rust
use std::thread::sleep;

use napi::bindgen_prelude::*;
use napi::Task;

struct DelaySum(u32, u32);

#[napi]
impl Task for DelaySum {
  type Output = u32;
  type JsValue = u32;

  fn compute(&mut self) -> Result<Self::Output> {
    sleep(std::time::Duration::from_millis(100));
    Ok(self.0 + self.1)
  }

  fn resolve(&mut self, _env: napi::Env, output: Self::Output) -> Result<Self::JsValue> {
    Ok(output)
  }
}

#[napi]
fn with_abort_controller(a: u32, b: u32, signal: AbortSignal) -> AsyncTask<DelaySum> {
  AsyncTask::with_signal(DelaySum(a, b), signal)
}

// .d.ts

export function withAbortController(a: number, b: number, signal: AbortSignal): Promise<number>
```

```typescript
test('async task with abort controller', async (t) => {
  const ctrl = new AbortController()
  const promise = withAbortController(1, 2, ctrl.signal)
  try {
    ctrl.abort()
    await promise
    t.fail('Should throw AbortError')
  } catch (err: unknown) {
    t.is((err as Error).message, 'AbortError')
  }
})
```

示例：通过 ts_args_type 和 ts_return_type 重写 TypeScript 生成的类型

```rust
use napi::bindgen_prelude::{Object, Result};

#[napi(ts_args_type = "a: { foo: number }", ts_return_type = "string[]")]
fn ts_rename(a: Object) -> Result<Object> {
  a.get_property_names()
}
```

```typescript
export function tsRename(a: { foo: number }): string[]
```

Tokio 深度集成

```rust
use futures::prelude::*;
use napi::bindgen_prelude::*;
use tokio::fs;

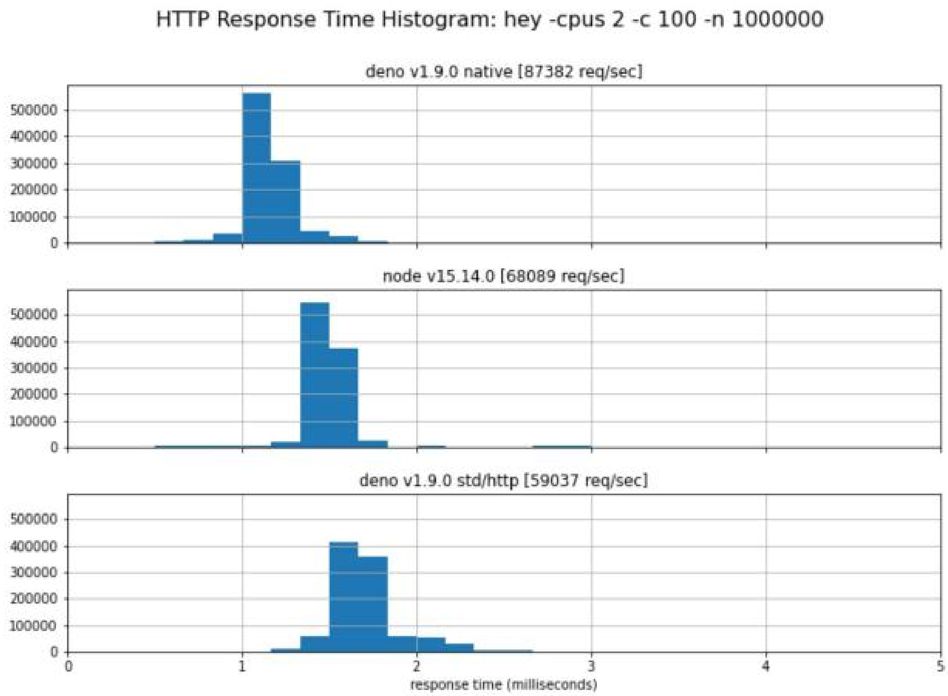
#[napi]
async fn read_file_async(path: String) -> Result<Buffer> {
  fs::read(path)
    .map(|r| match r {
      Ok(content) => Ok(content.into()),
      Err(e) => Err(Error::new(
        Status::GenericFailure,
        format!("failed to read file, {}", e),
      )),
    })
    .await
}
```

- 复用 Tokio 生态，比如可以像 deno 那样将 http server 层用 Rust 实现而不是用 JavaScript 实现；
- 可以受益于 Tokio 强大的工具链，比如 tokio-console 与 tokio-tracing；

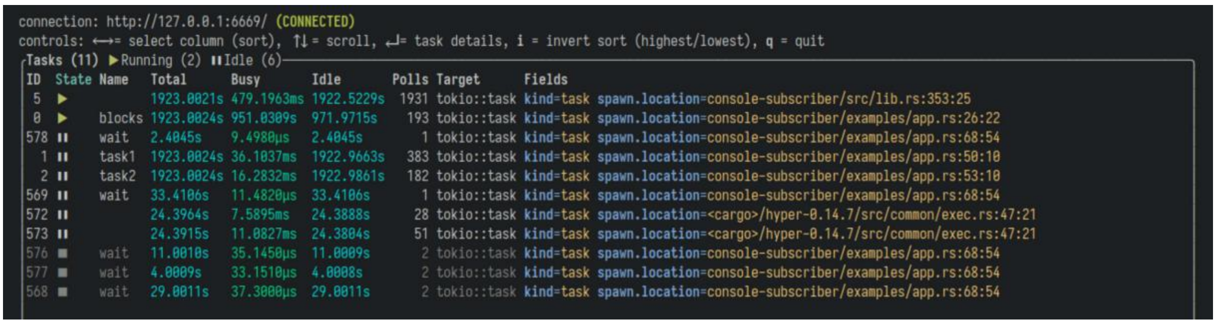
- 提前享受 io_uring 等 libuv 未封装的内核新特性；

参考网址：https://github.com/Broooooklyn/hns

下图展示 Deno 将 HTTP serverJS 实现转为 Rust 实现后性能的前后对比，最下面的图是替换前，最上面的图是替换后，QPS 提升 30%+，已经超过 Node.js。

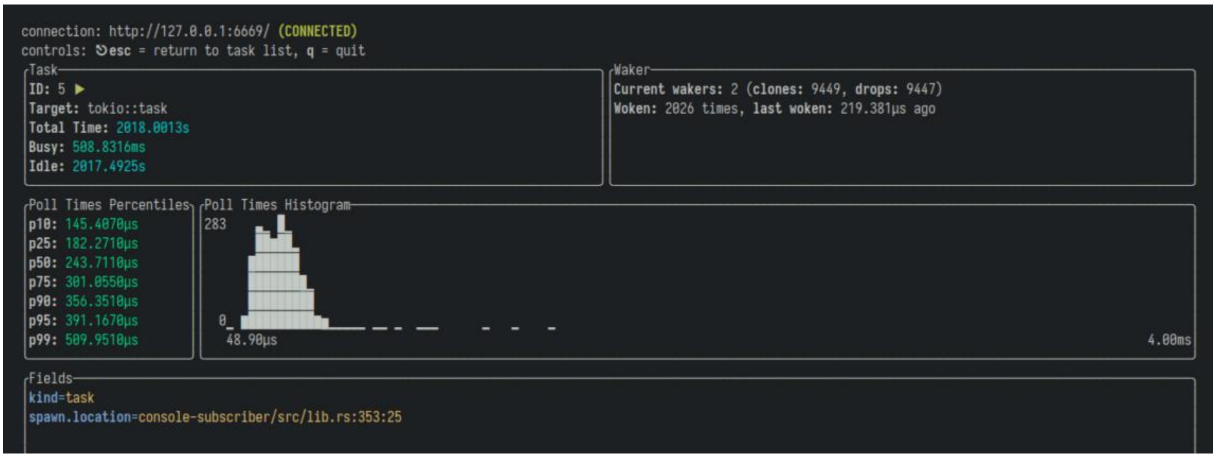


与 Tokio 深度集成后，可以享受 Tokio 生态中的先进工具，如 Tokio console（下图），在框架系统提供一个类似任务管理器，可以看到任务信息。



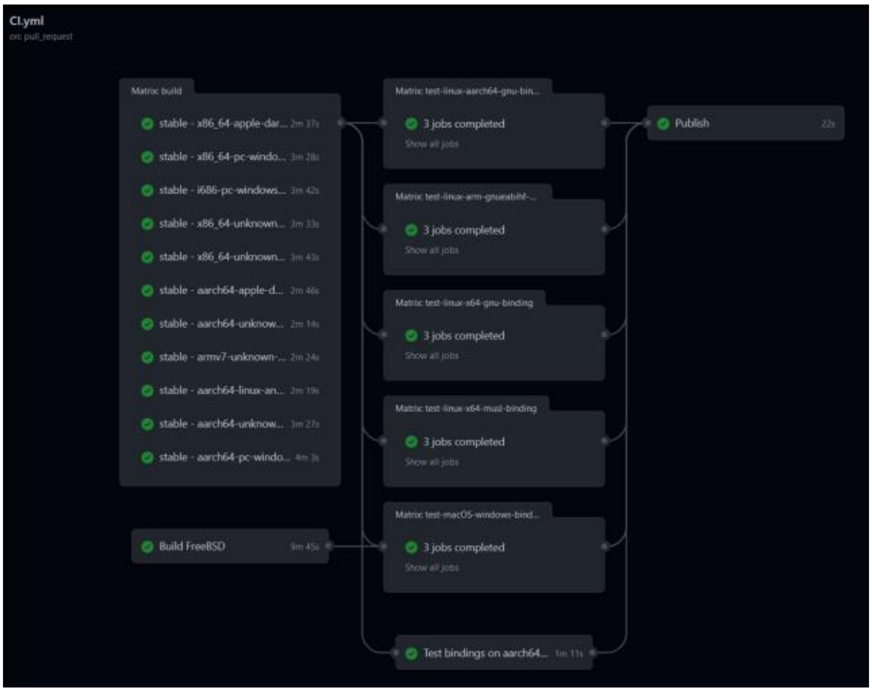
参考网址：https://github.com/tokio-rs/console

Tokio console 的另一个功能，统计任务的耗时、运行时间、等待时间等细致的信息。



开箱即用的 CLI

- 新建项目与 GitHub Actions 模板，囊括跨平台编译与发布；
- 便捷的 build 命令，预置各种与 Node.js Addons 相关的编译参数，使用便捷；
- optionalDependencies 版本号管理，发布命令，GitHub CI Artifacts 命名与移动等功能全部封装在内，开发者无需关注底层繁琐的实现细节与维护。



GitHub Action 的执行效果

下图列举了 NAPI-RS 支持的操作系统、CPU 和环境，以及 Node 版本。

Support matrix

	node12	node14	node16
Windows x64	✓	✓	✓
Windows x32	✓	✓	✓
Windows arm64	✓	✓	✓
macOS x64	✓	✓	✓
macOS arm64	✓	✓	✓
Linux x64 gnu	✓	✓	✓
Linux x64 musl	✓	✓	✓
Linux arm gnu	✓	✓	✓
Linux arm64 gnu	✓	✓	✓
Linux arm64 musl	✓	✓	✓
Android arm64	✓	✓	✓
FreeBSD x64	✓	✓	✓

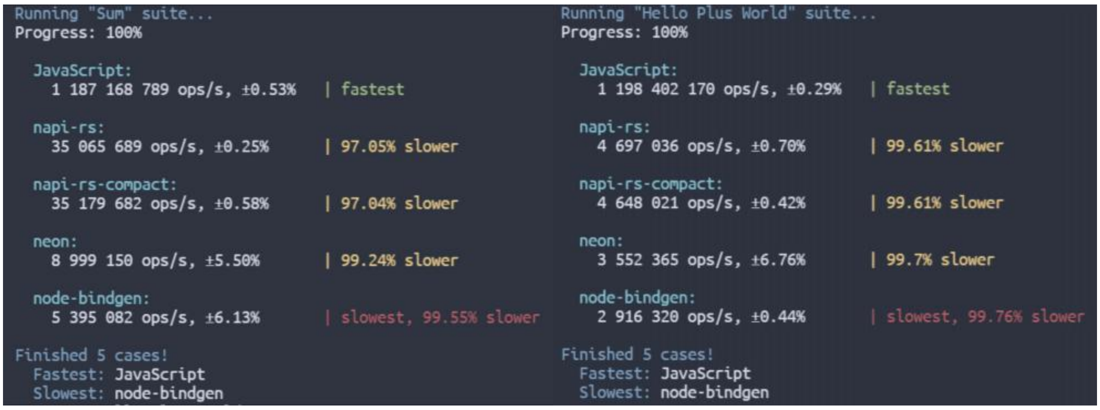
参考网址：https://napi.rs/introduction/getting-started

2、使用场景

- 性能敏感的计算场景，输入输出简单；
- 需要使用 Node.js 不支持的 Native 特性，比如特定的 CPU 指令集，调用 GPU、USB ，调用系统 API 等；
- 在 Server 端共享跨语言与框架的代码，比如 Prisma 使用 Rust 开发核心的 query engine，再封装到 Node.js Go 等；
- 利用 Rust 现代化的工具链(包管理、跨平台编译等)粘合现有的 C++ 代码，比如 https://github.com/Brooooooklyn/canvas

调用开销

下面左图是使用 JavaScript、NAPI-RS、neon、Node-bindgen 的调用开销对比，右图是函数字符串 Hello Plus World 的调用开销示例。



@node-rs/crc32

从下图示例看到 NAPI-RS 封装后的 crc32 比现有的 C++包以及 JS 实现的速度快。



参考网址：https://github.com/napi-rs/node-rs/tree/main/packages/crc32

各种类型的 hash 计算

下图示例是截取自字节跳动 Node.js info 包，Rust+NAPI-RS 比 JS 计算逻辑快 20 倍以上。

```
// Node.js trace hash function
// From ByteDance Node.js infra
// 20x faster than BigInt calculate logic in Pure JS

#[napi]
fn log_id_hash(input: String) -> u64 {
  let hash = fnv(input.as_bytes(), FNV1_64_INIT);
  linux_core_hash_u64(hash)
}
```

node-pty

- 封装系统调用；
- 实现跨平台；
- 预编译；

参考网址：https://github.com/microsoft/node-pty/issues/507

@napi-rs/canvas

- 将 C++ 编写的 Skia 项目编译成 static link library，通过 Rust FFI 机制引入使用；
- 组合现有的 Rust crates 比如 cssparser，base64，avif 等；
- 用 NAPI-RS 封装成 npm 包，对比 node-canvas 0 系统依赖，支持更多系统与 CPU 架构，功能更多。

下图是 napi-rs/canvas 的执行效果，对比 Node canvas 多了一些功能，比如：彩色字体支持，不需要系统依赖。

```javascript
const { writeFileSync } = require('fs')
const { join } = require('path')

const { createCanvas, GlobalFonts } = require('@napi-rs/canvas')

GlobalFonts.registerFromPath(join(__dirname, 'fonts',
'AppleColorEmoji@2x.ttf'), 'Apple Emoji')
GlobalFonts.registerFromPath(join(__dirname, 'fonts', 'COLRv1.ttf'),
'COLRv1')

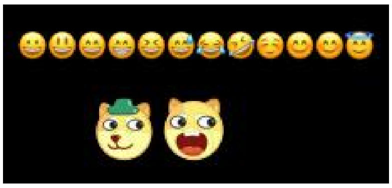
const canvas = createCanvas(760, 360)
const ctx = canvas.getContext('2d')

ctx.font = '50px Apple Emoji'
ctx.strokeText('😀😃😄😁😆😅😂🤣☺️😊😊😇', 50, 150)

ctx.font = '100px COLRv1'
ctx.fillText('abc', 50, 300)

const b = canvas.toBuffer('image/png')

writeFileSync(join(__dirname, 'draw-emoji.png'), b)
```



参考网址：https://github.com/Brooooooklyn/canvas

3、与 Neon/node-bindgen 对比

- NAPI-RS 有更好的性能/更小的调用开销；
- NAPI-RS 提供从开发到发布的全流程解决方案，而不止是仅仅在 Rust 层提供 Node-API；
- NAPI-RS 对比 Neon 提供的 Node-API 更全面，包含了所有 Node-API 的 C API 封装；
- NAPI-RS 对比 node-bindgen 提供可选的 napi 版本选择，node-bindgen 只能支持最高版本的 Node.js，而 NAPI-RS 可以通过 features 来选择支持的 Node.js 版本，最低支持到 10.0.0；
- NAPI-RS 与 node-bindgen 的异步运行时不同，NAPI-RS 支持的是 Tokio，而 node-bindgen 支持的是 fluvio。

参考网址：https://napi.rs/neon

四、Rust 与 Node 的未来展望

- Rust 是 JavaScript 基建的未来，Next.js、Parcel、Prisma、Rome 仅仅是革命的开始；

- 随着生态的丰富，未来用 Rust 来编写 Native Addon 会更加便捷，各种配套的基础建设会逐步丰富；
- 除了通过 Node-API ，未来 Rust 会有更多形式为 JavaScript 生态提供服务，比如 Rome 打算嵌入 JavaScript 引擎，并提供简单的 JavaScript API，而不是依托于 Node-API，比如 Deno 正在实现–compact 模式（参考链接：https://github.com/denoland/deno/issues/12577），届时通过 Deno + Deno FFI + npm 包也能启动存量的 Node.js 项目。(NAPI-RS 已经计划通过 feature flag 支持无缝编译到 Deno FFI)。NAPI-RS 还计划提供无缝编译到 wasm，让包的发布和维护更简单；
- 越来越多的前端与 Node.js 业务相关的代码会出现在 Rust 的 crates 中，各种自研渲染引擎，比如小程序和类 Flutter 方案，更容易使用 Rust 开发。以后可能会诞生基于 Rust 生态的集运行时、构建、开发调试工具等周边的一体化解决方案。

前端体验

基于大数据的前端页面异常结果检测

摘要：在 Web 监控领域内，对于页面在实际用户场景下，每次展现的内容是否符合预期，是我们建立监控体系的一个最根本诉求。基于这个根本的诉求，我们建立了诸如资源下发、脚本执行、性能埋点等多种维度的监控。但面对越来越复杂的业务逻辑，千人千面的产品设计和个性化功能，一个页面是否停在白屏、骨架屏、错误页、或模块丢失仅渲染一半内容等“非预期的页面展现结果”而造成的体验欠佳，则始终缺乏直接且有效的手段来进行监控，本次介绍一种较新的基于数据检测的方式用于前端页面结果的监控。

分享人：王骕(肃马)，阿里巴巴前端技术专家，淘系前端技术专家，多年 B 端数据产品研发经验。

目录：

一、研发背景
二、观测
三、检测
四、适用性及其它
五、工程化及产品化

最早提出的问题是一个页面投到客户端之后白屏率实际有多少？就是纯白的情况，后来反馈过来还是有问题。白屏可能不是我们最终的检测目的。实际上是一些有问题的页面。

一、研发背景

定义问题：如何在客户场景下发现非预期的页面展现结果？

- 页面在实际用户场景中展现内容及交互过程是否符合预期，是建立监控体系的最根本诉求；

- 监控中所称“白屏”，是非预期结果中的一种极端情况，其检测条件是最容易简化的，但并非直接目的；
- 不能直接观测，是解题困难的根本原因；



从左到右，罗列了 6 种常见的异常情况：

- 第一张页面是带背景色的纯空页面；
- 第二张是有一骨架屏，JS 没有执行；
- 第三张是结果没有返回，但是页面的功能正常，动作都触发了，结果没回来；
- 第四张是常见的错误页，逻辑错误实在没法执行了，页面提示用户刷新或者反馈；
- 第五张是一个安全的拦截，大多数情况都不是研发主动去写的，使用了第三方的包，检测到有异常的情况下显示；
- 第六张是一个比较复杂的情况，上半截正常，下半截内容没显示出来的原因有很多。

这六种情况罗列出来，如果只讨论白屏的话，只有第一种情况勉强算，后边的没办法定义它的状态。我们要做的就是把这六种情况都检测出来。

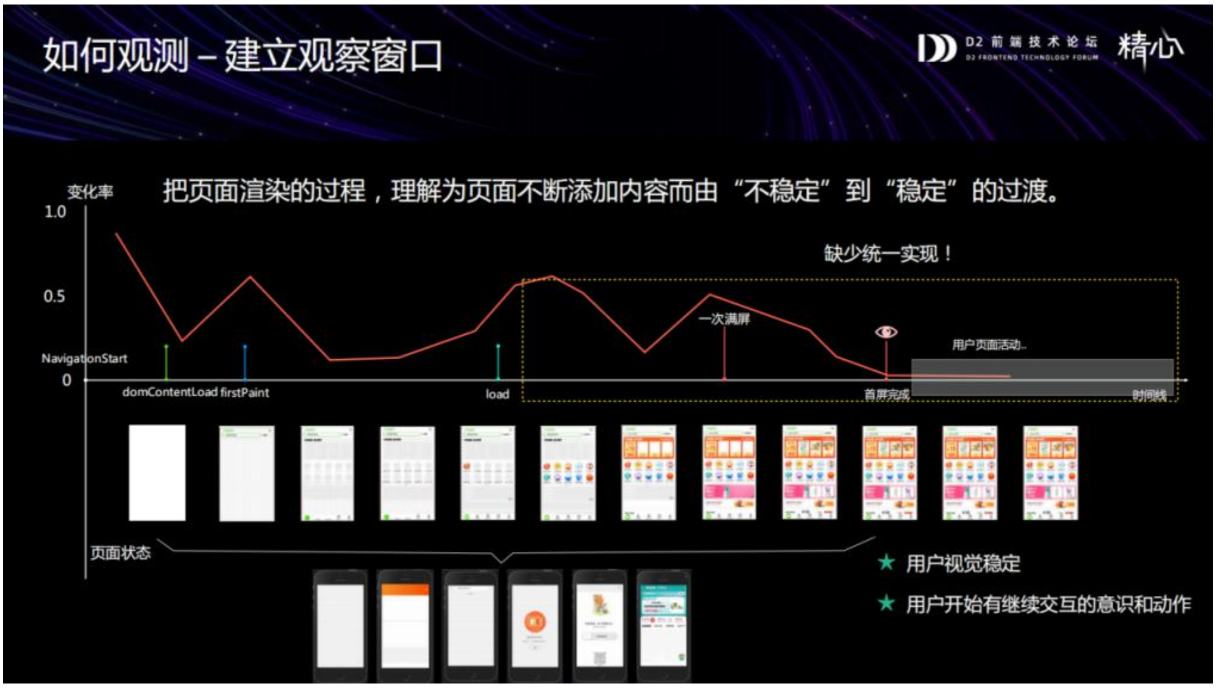
页面在实际场景中展示的是否和预期一致，这是我们建立监控体系的一个根本诉求。监控里所讲的白屏，其实是这些所有的异常中最极端的一种情况。这种情况并非我们检测的直接目的。

我们监控的直接目的是尽可能的都把问题发现。

做检测很困难的原因在于不能够直接的去观测，只能间接地去观测，这是导致问题困难的根本原因。比如：一个页面上中下三部分，上边轮播，中间正文主题，下边列表。实际展示并不是这样，中间的任何一个模块出问题，都会间接的导致最后显示有问题。

二、观测

1、如何观测 - 建立观察窗口



看上图中的这个页面，是从最开始的渲染一直到整个页面渲染完毕的整个过程，我们可以把页面渲染的过程，理解为页面不断添加内容而由“不稳定”到“稳定”的过渡。

图中左边用户开始访问 NavigationStart 的时间点是 0，接着是一些常见的事件，然后有一个独特的时间点，就是首屏完成，用户的视觉稳定，从这个点开始，用户才开始有继续交互的意识和动作。

我们发现，前边定义的六个异常状况，其实都是发生在首屏完成之前的。

如何做首屏完成的时间点？业界并没有统一的标准。这是一个技术指标，不是一个用户体验指标，完整用户体验指标还是在首屏完成后的位置。

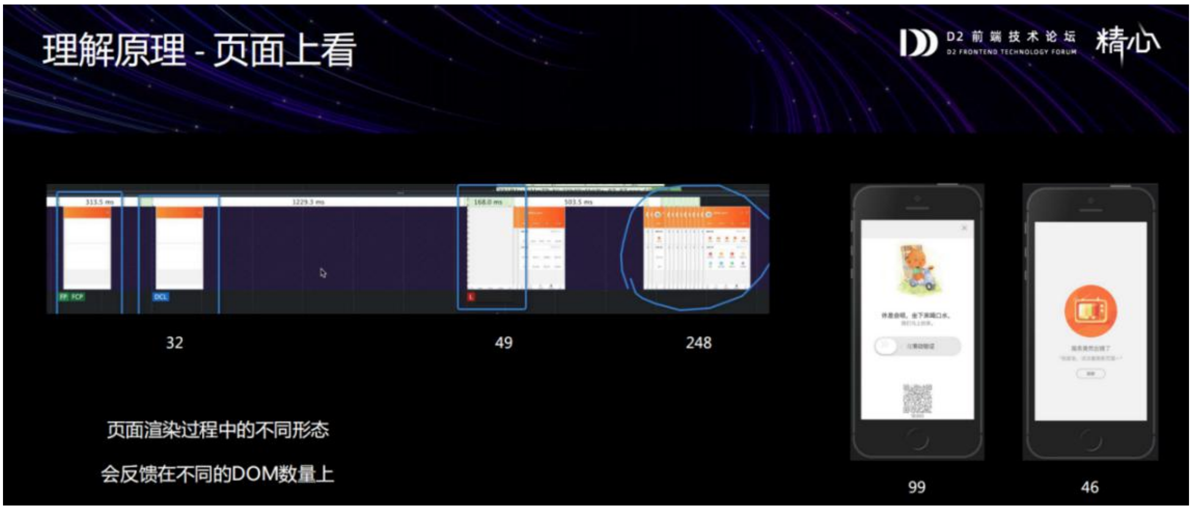
如何检测完整用户体验指标这个点?

页面的渲染过程，是由不稳定到稳定的一个过渡，首屏稳定了也就是页面完成了。我们要把多余和缺失显示的统计出来，然后去除以面积就是变化率。变化率会不断波动，不管如何波动，最后都是趋近于零的，也有可能不完全是零，因为可能有轮播、倒计时等在运行，但会有一个周期性相对稳定的波动。监测到这个点就拿到了首屏时间。

做了这个观测窗，第一个解决了性能问题，首屏加载时间，第二个解决了在点位上可以打指标，比如说图片数，节点数，这个节点打到服务端，剩下的问题就是如何检测。

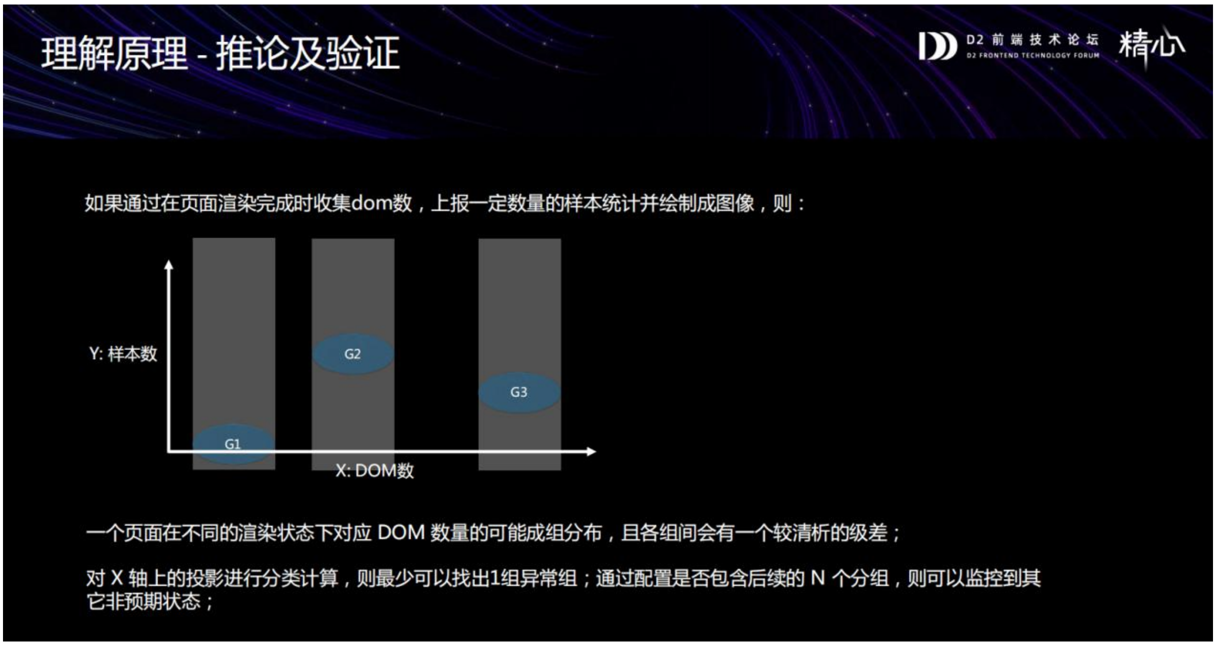
三、检测

1、理解原理 - 页面上看



我们从上图中这个页面看，从左到右。所用指标就是节点数。页面渲染中不同的形态会反馈在 DOM 数量上，大部分的检测都是通过 DOM 树来判断的，如何去确定一个稳定的 DOM 树来做出判断?

2、理解原理 - 推论及验证



先做一个推理，如果在首屏渲染完成的时候，收集 DOM 树，上报到服务端，绘制成图像，我们能不能达到这两个状态：

- 一个页面在不同的渲染状态下对应 DOM 数量的可能成组分布，并非是点，该页面的 DOM 数量一定是浮动的，不会太大，且各组间会有一个较清析的级差，就能够清晰的识别出这个页面到底有多少种状态；
- 对 X 轴上的投影进行分类计算，则最少可以找出 1 组异常组；通过配置是否包含后续的 N 个分组，则可以监控到其它非预期状态。

3、验证推理 - 统计上看

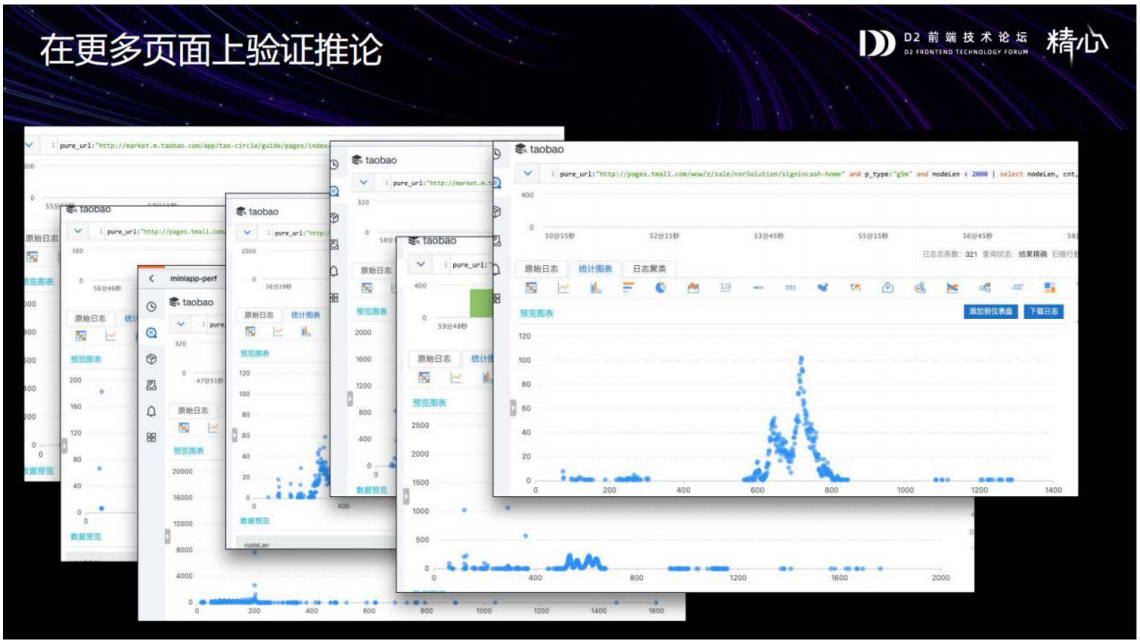


有了推理之后，我们来验证这个推理。依旧是这个页面，已经记录了 DOM 树的数量，放在用户场景中，对应 DOM 数量验证。

4、在同一页面的不同时间上验证

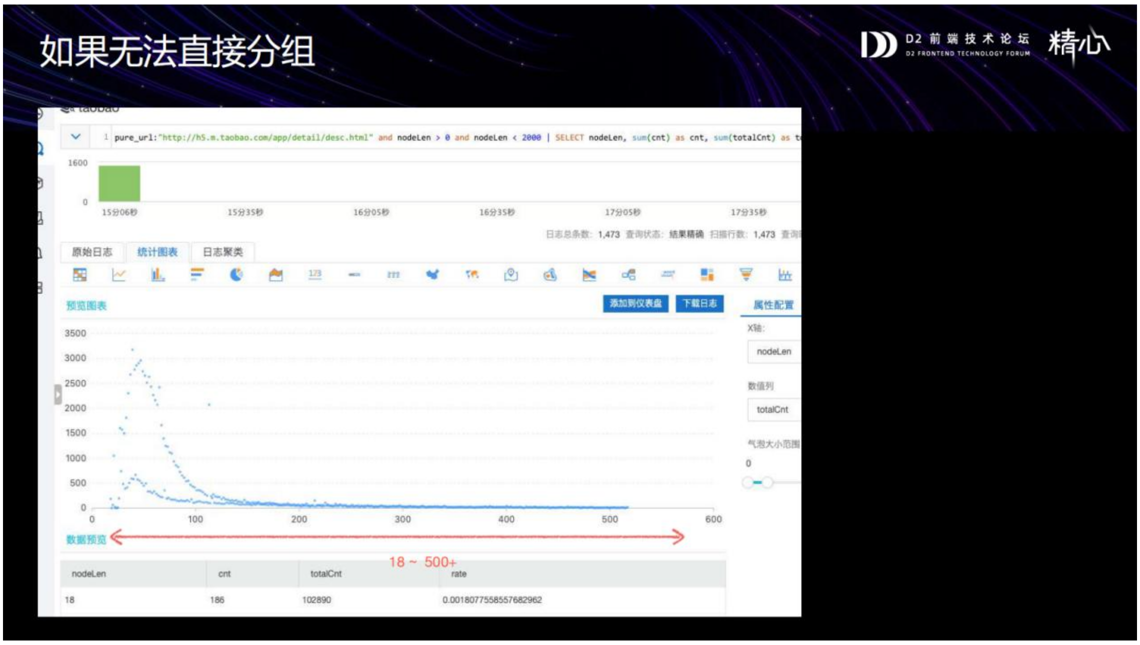


5、在更多页面上验证推论

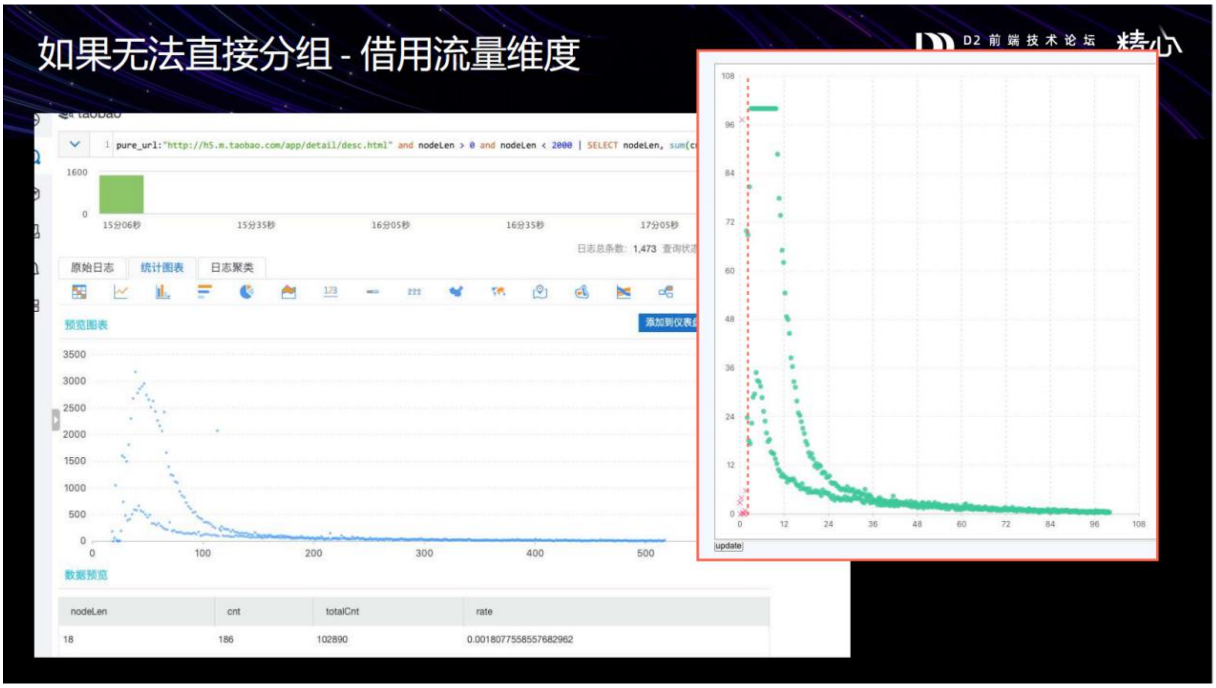


在大量的页面上加以验证，我们才可以使用此逻辑。最左边是否有分组，有分组即是正确的；时间上是否稳定，稳定即正确，同时满足即可。

6、如果无法直接分组 - 借用流量维度







7、总结原理

- 大部份页面在不同的渲染状态下对应 DOM 数量的可以成组分布，且各组间会有一个较清析的级差；

- 少部分不能直接分类的页面中，由于异常点集中于左下，所以仍可以根据其它方法进行样本分类，其结果不具备绝对性，但比例变化时仍具参考性；
- 页面在客户侧的渲染形态分布是基本稳定的，如果渲染状态发生了变化那么由一定对应服务端或前端功能的改变（功能或策略变动，页面形态切换，故障等）。

四、适用性及其它

1、适用性范围

- 纯静态及 SSR 页面；

二者都是从服务器端推下来，不管脚本是否执行，页面都不会空，而且都是和现实的内容一致。这种情况下，DOM 树去判断没有意义；

- 互动游戏类，SPA 页面；

互动的特点是 canvas 画出来的，其它的什么也没有，这种的波动范围非常小，没有办法做分组，跟踪不到具体哪一组异常，不适合做监控。

2、结果比较偏高的原因

- 重定向、拉端跳出；

比如：A 页面跳到 B 页面，或者说 A 页面跳到 B 应用，A 页面不是服务端的重定向，是 JS 的 location 重定向，是业务脚本里边主动的，上报的样本是缺损的样本，端外拉端的比例比较高；

- 低速网络；

例如进地铁或电梯，网络不好，超时，上报的样本就是缺损的样本。

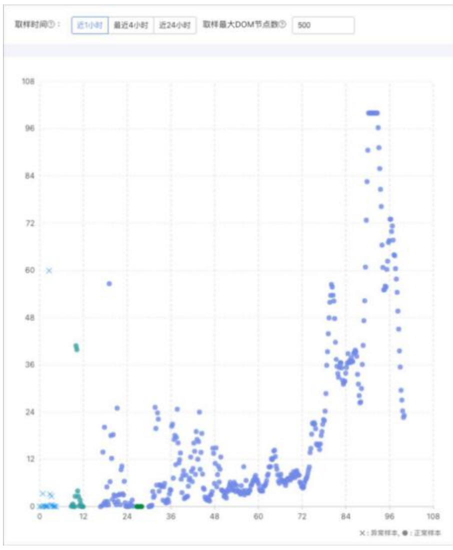
3、未来研究方向

用户持续交互过程中的缺损情况：首屏以外、交互过程中、Tab 切换、下滑载入、翻页等；

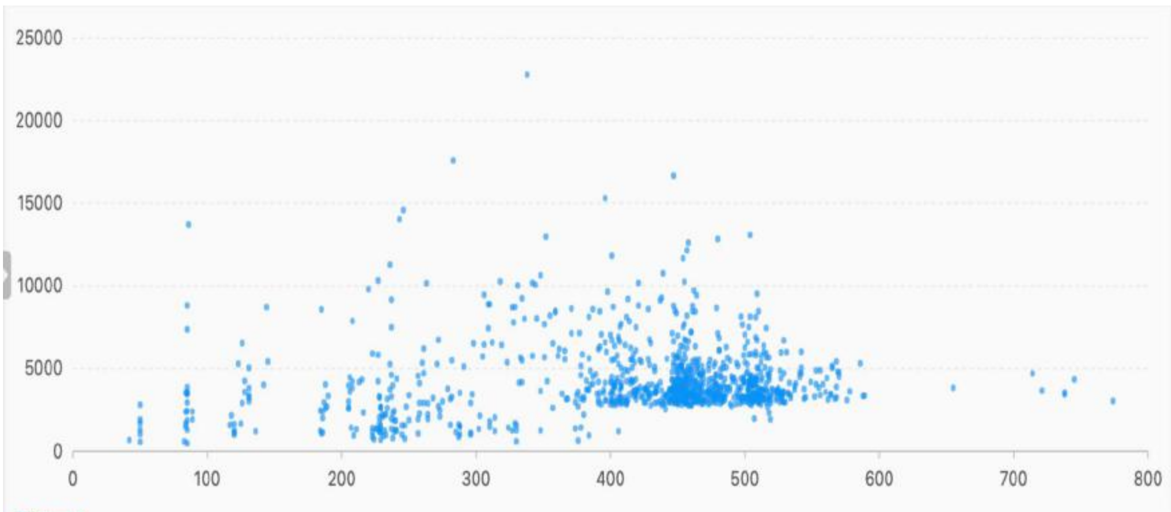
缺损位置定位，小样本定位。

五、工程化及产品化

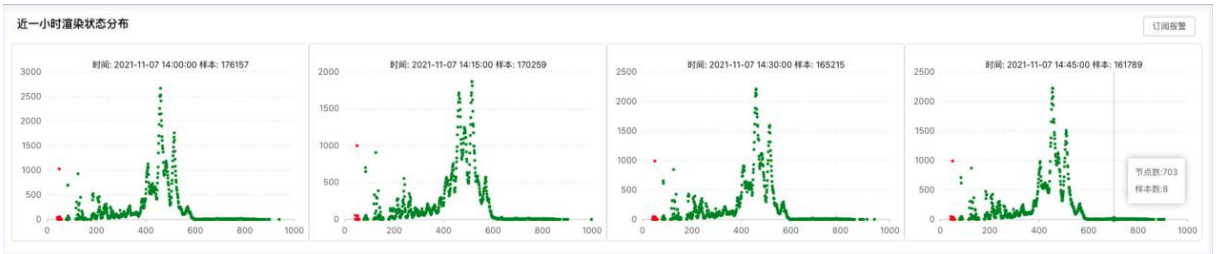
工程化大部分和前端不相关，逻辑较为复杂，细讲繁而杂，这里就做个简化讲述。产品是如何搭建起来的？能够透露出一些什么问题？



上面这张图是突出最频繁的，且整理后已经规范了的，他的前身是下图：



截图的窗口是首屏，采样的是首屏完成的时间和 DOM 的节点数。横轴是 DOM 数量，纵轴不是样本数量而是首屏完成的时间。实际计算时，首屏什么时候完成的与页面是否异常关系并不大，即便效率慢，也一步步完成了，所以时间维度不做指标，简化而来了此图。



同样一张图，基于统计，对应的时间窗口，15 分钟一张图，排列出来。

还有一个就是变化率，当前这个时间有多少个变化的点。



上图是想找一些有异常状态的页面，但大部分都是贴底的、波动特别小的，状态特别稳定的页面，并不能看到什么问题。正好此页面有爬虫，研发人员打开了安全拦截，所以可以清晰的看出爬虫工作的时间段和数量，能够检测出问题的状态。



用户行为链路的识别、挖掘及诊断

摘要：在产品体验优化的过程中，需要不断深入分析和挖掘用户在产品使用中的体验问题。行为分析是体验优化中最核心的部分，而链路分析是行为分析中最重要的一环。这一次分享首先聊一聊行为链路的分析理念，其中包含了全局行为链路挖掘、指定链路分析、链路下钻分析、链路关联分析、链路多维分析等。其中全局行为链路挖掘最为关键，它是用户行为链路分析的起点，可以在纷繁复杂的行为数据中找到关键路径及问题路径，串联更深入的分析。我们探索性的将行为链路智能识别的能力应用到行为分析当中，通过整合趋势路径、关注热点路径来收敛链路，并利用在实际业务中沉淀的诊断模型，智能诊断链路，发现产品链路中的体验问题，为产品体验优化提供决策依据。 本次分享我们将介绍我们在探索行为链路分析背后的理念、方案和创新点。包含：行为链路识别算法，复杂链路可视化方案，链路诊断模型，以及实际的业务落地案例等。

分享人：徐昊(南冬)，阿里巴巴前端技术专家，阿里巴巴内部体验管理平台 AEM 的行为分析方向负责人。

目录：

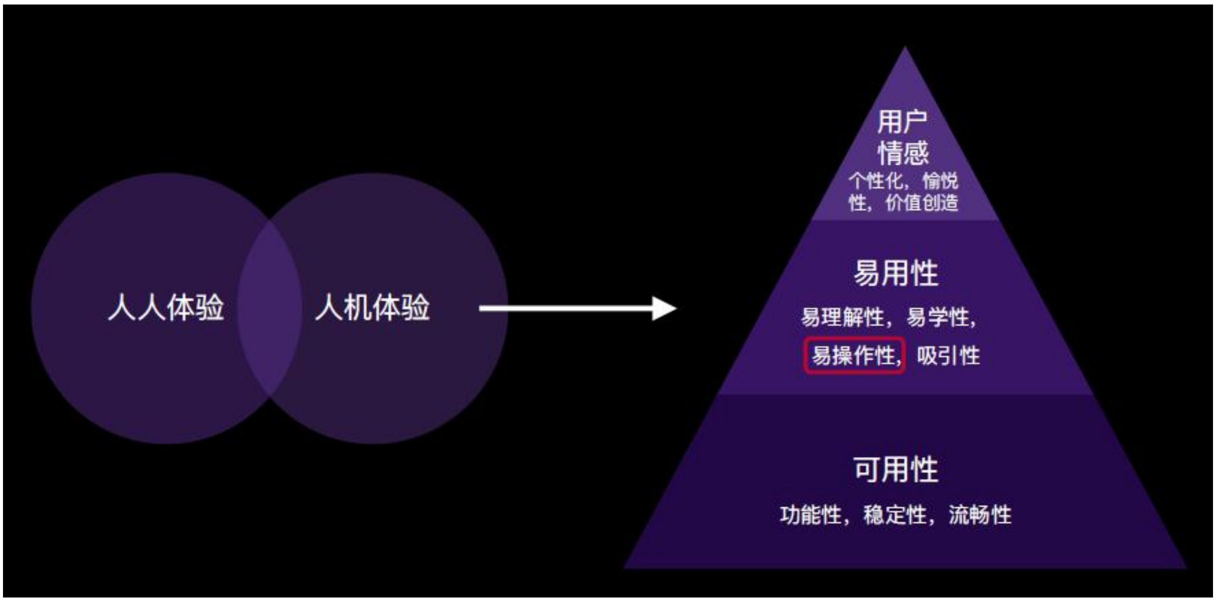
一、案例篇（背景知识、案例分析），

二、技术篇（链路识别、链路可视化、链路诊断、未来展望）

本篇内容将通过案例篇、技术篇来介绍行为链路的识别、挖掘及诊断。

一、第一部分：案例篇

1、背景知识



1）用户体验

- 首先我们把用户体验分为两个部分：人人体验和人机体验：
  - 人人体验：就是人与人之间提供服务时所产生的体验。
  - 人机体验：就是用户用我们的产品中进行人际交互的时候所产生的一些体验；

- 根据国家的标准和我们理解，人机体验分为三个层次：
  - 可用性：包括（功能性、稳定性和流畅性）；
  - 易用性：包括（易理解性、易学性、易操作性、吸引性）；
  - 用户情感：包括（个性化、愉悦性、价值创造）；

从可用性上来讲，目前比较成熟，相应的有稳定性和流畅性的指标；但是在易用性上目前比较薄弱，一般是由人的主观感知来衡量这个产品，它到底好不好用；对于易用性，我们要用研究用户行为来解决。那本次分享呢，也是聚焦在易用性的易操作性。

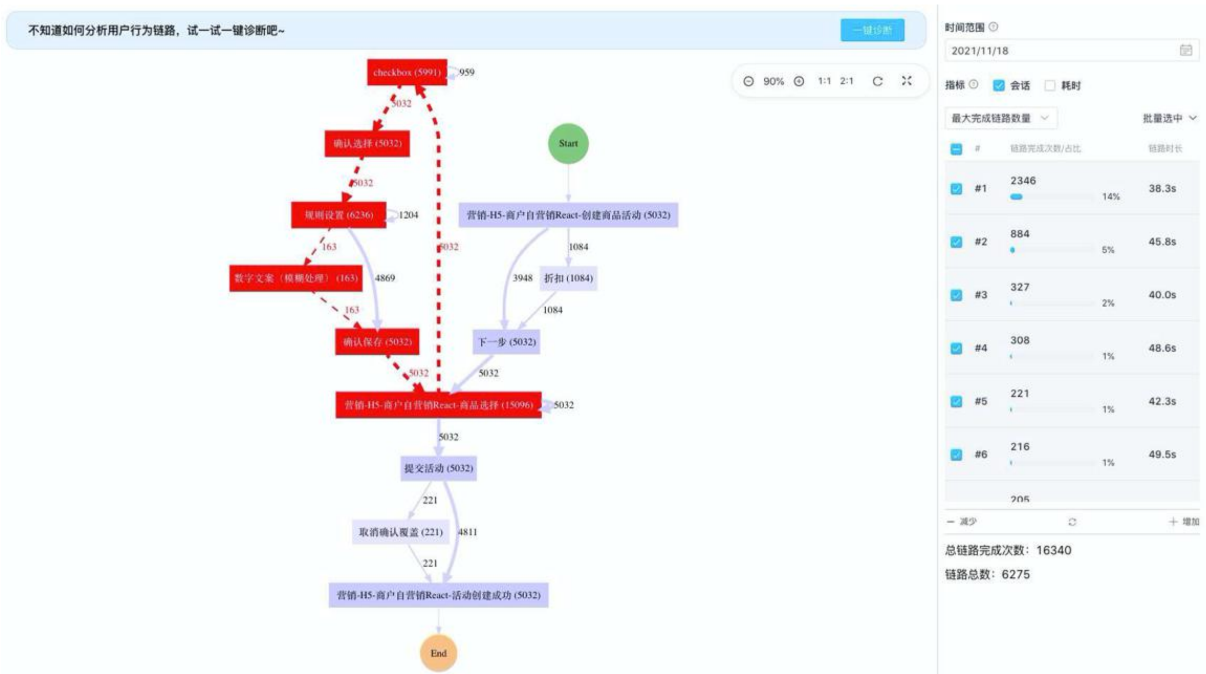
2）用户行为



用户行为是由访问、浏览、操作这三大构成的，覆盖了用户在我们产品中所产生的所有行为。用户使用我们人机交互过程中产生的数据，称之为用户行为数据。

如何对用户行为的数据进行分析呢？我们在过去的这个分析案例和场景中，经过思考和沉淀，得出了一个用户行为分析模式，它分为四个部分：

访问链路分析→用户链路分析→指定链路分析→操作下钻分析



- 首先是访问链路分析，从全局页面的维度去看整个用户的这个访问链路情况，我们可以在其中发现一些流量和流向与我们产品最初设计不符的地方；
- 然后从页面的全局操作维度去查看用户的这个真实操作路径，并通过智能诊断发现用户的操作链路上的一些关键路径的问题路径；
- 那基于我们这些问题路径、关系路径，进一步的分析流失和操作失常这些指标来量化；
- 假如有两个节点流失过多，我们还可以看一下下端，分析出，这两点间的一个具体操作路径。

经过层层递进的四步分析模式，可以去定位产品上操作体验的问题，并下载找原因，同时优化结束后，还可以用这套模式来衡量优化的效果。

3）案例分析

用户反馈体验差



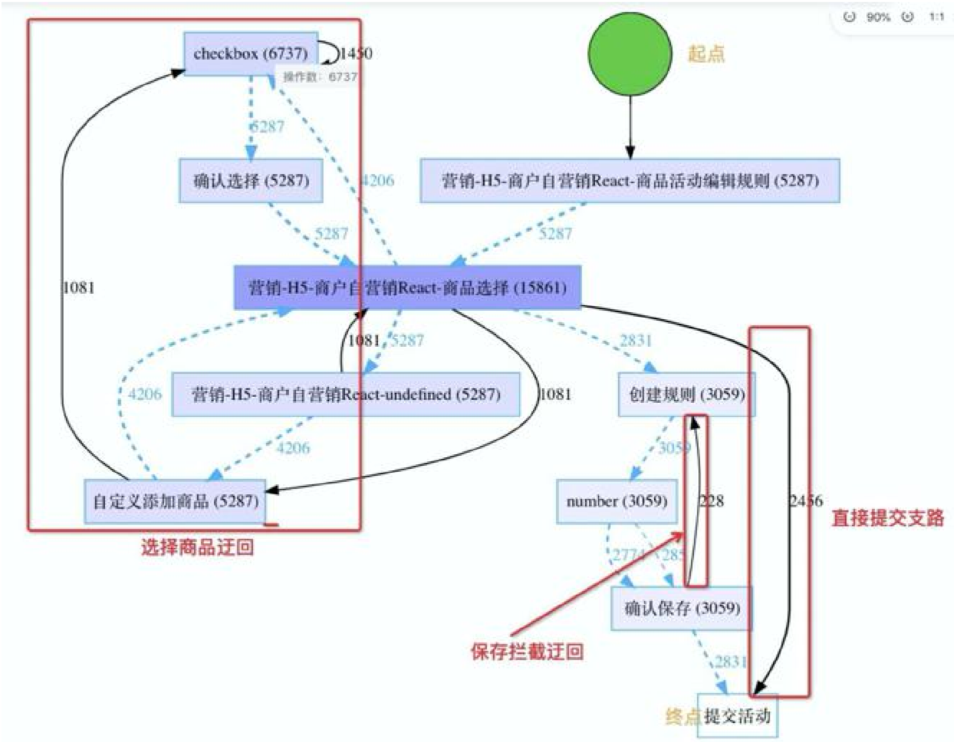
这是外卖商户在后台进行商品活动配置的一个页面。

商品活动配置是我们外卖营销中一个很重要的工具，大家点外卖的时候看到的推荐商品，就是通过这个工具来配置出的。经过我们对商户调研，收到很多用户反馈，大多都集中在繁琐、不好用、不方便等。

如何改版

那对于下一步如果改版，我们内部也展开了一些讨论，但是我们可以发现，这里面包含假设、可能，从场景组合和个人主观认知出发，很难得到结论，不清楚用户真实的操作习惯，迭代没有数据支撑。

案例应用 - 问题挖掘



用户操作链路



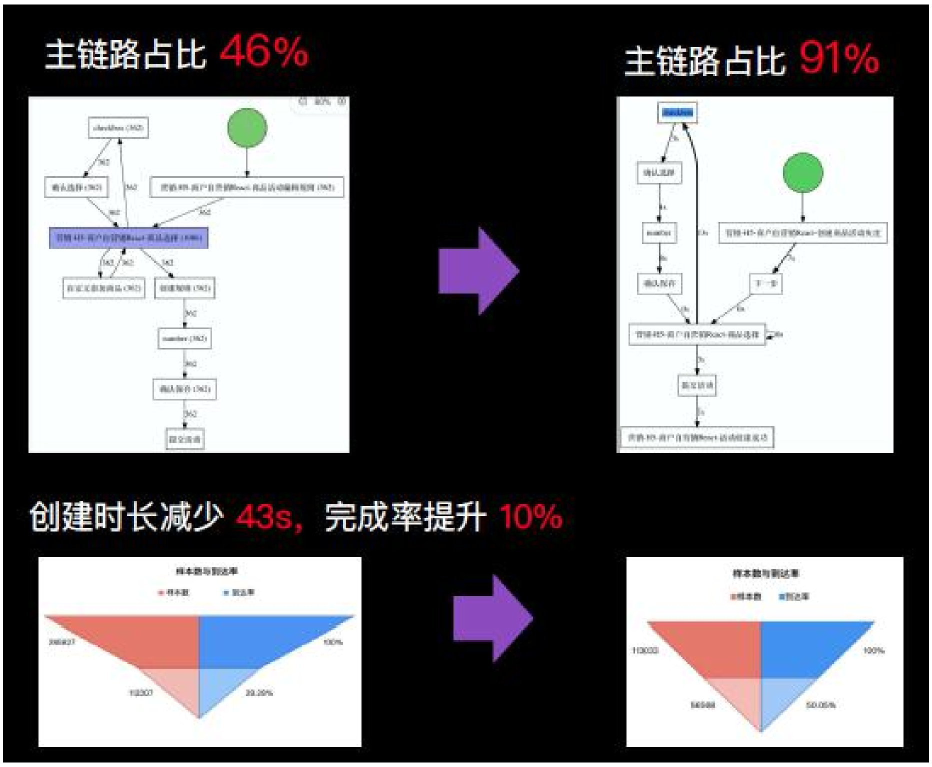
提交链路分析



下钻分析

首先从用户链路分析中，我们可以发现用户操作链上的一些问题，包括迂回操作，操作拦截，操作支路等。经过下钻分析，我们发现 90%的用户都是操作创建一个规则，一个商品，我们之前纠结的批量创建好，还是单个创建更好，就有了答案，为什么用户觉得不好用呢？因为我们的交互是以批量的角度去设计的，违背了用户的一个操作习惯。

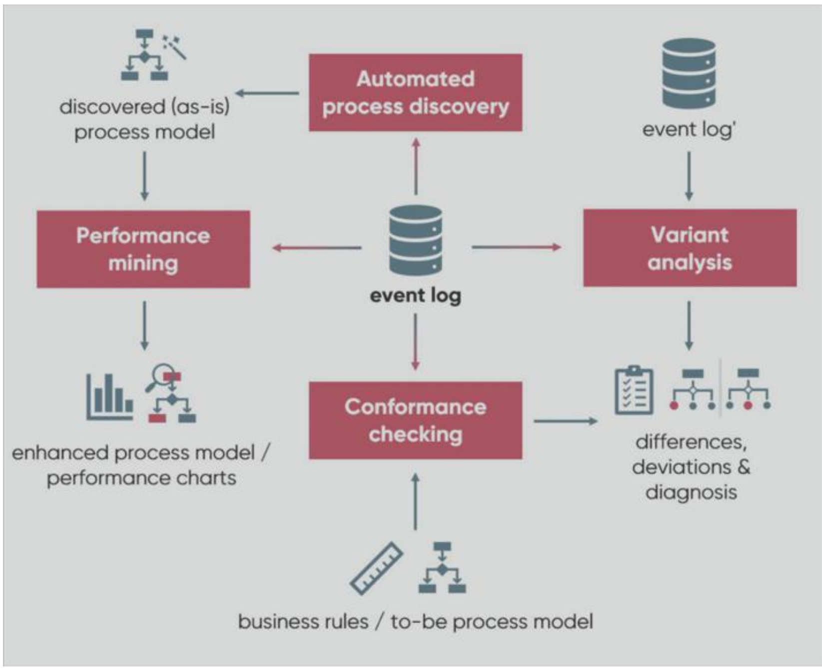
案例应用 - 效果验证



通过对问题的发现和用户真实性的感知，我们也做了一些优化组合，上线后从客观数据上产生提升，用户的一个总操作链路提升很多，到了 91%，说明用户的操作更符合了我们产品设计的整体链路，整体的转化，也上升了十个点。除了数据上的提升，我认为另一部分价值是产品的

交互的设计，不再仅仅依靠人的主观经验，而是能够根据用户的真实操作习惯来去设计，从而提升我们产品的用户体验。

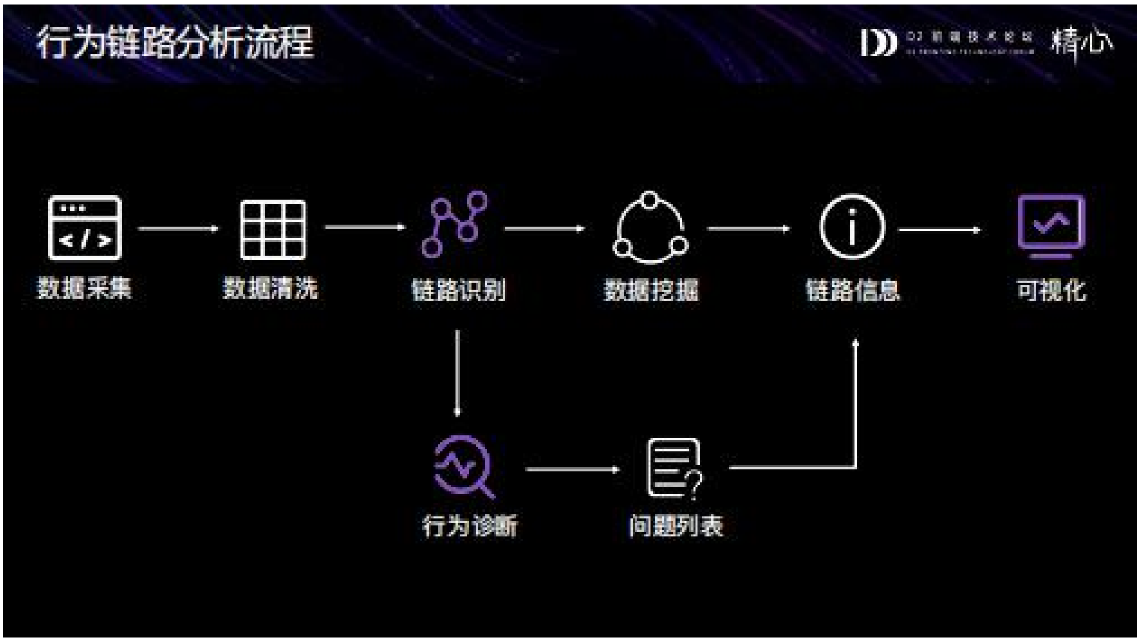
二、第二部分：技术篇



1、概念

1）流程挖掘

流程挖掘并不是一个新的概念，维基百科上面给流程挖掘定义，就是通过专门的数据挖掘算法，来识别系统的日志，最终得出数据中包含的信息、趋势、模式。这次是创新的把流程挖掘的思想跟行为分析相结合，将用户行为看作成瞬时发生的业务流程去分析。而他们最大的不同是，业务流程是已知且确定，而用户操作链路是未知且非确定。



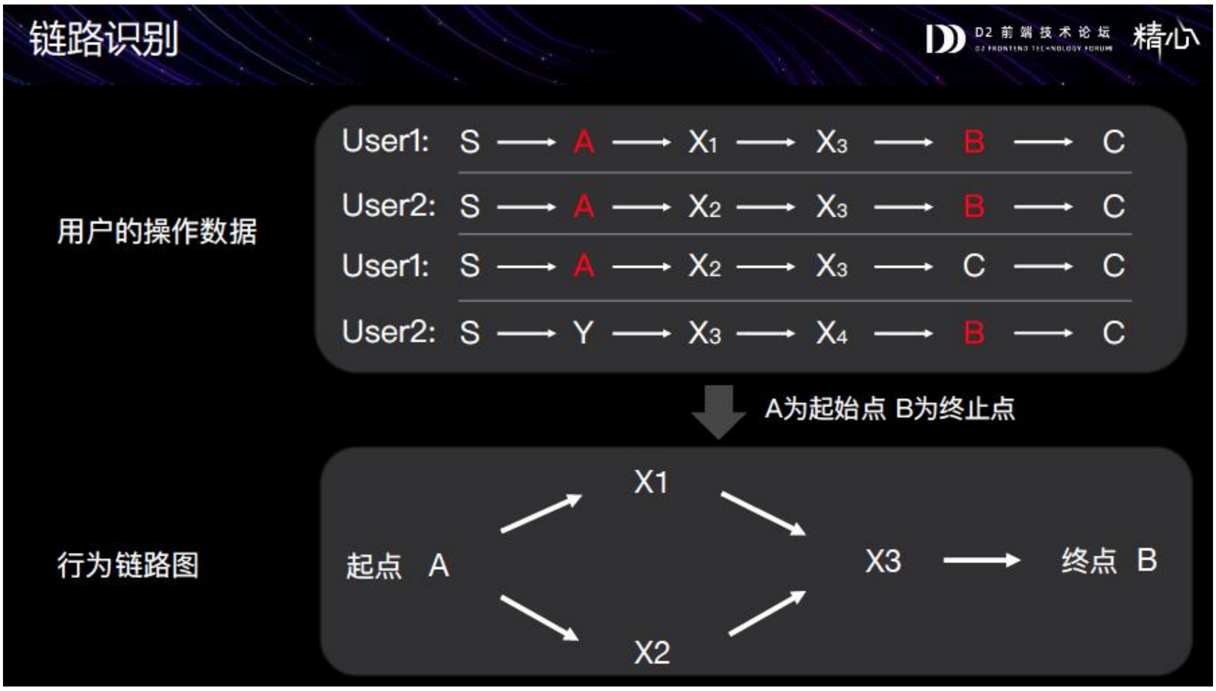
行为链路的分析流程

2）行为链路分析流程

行为链路的分析流程大致包括以下部分：

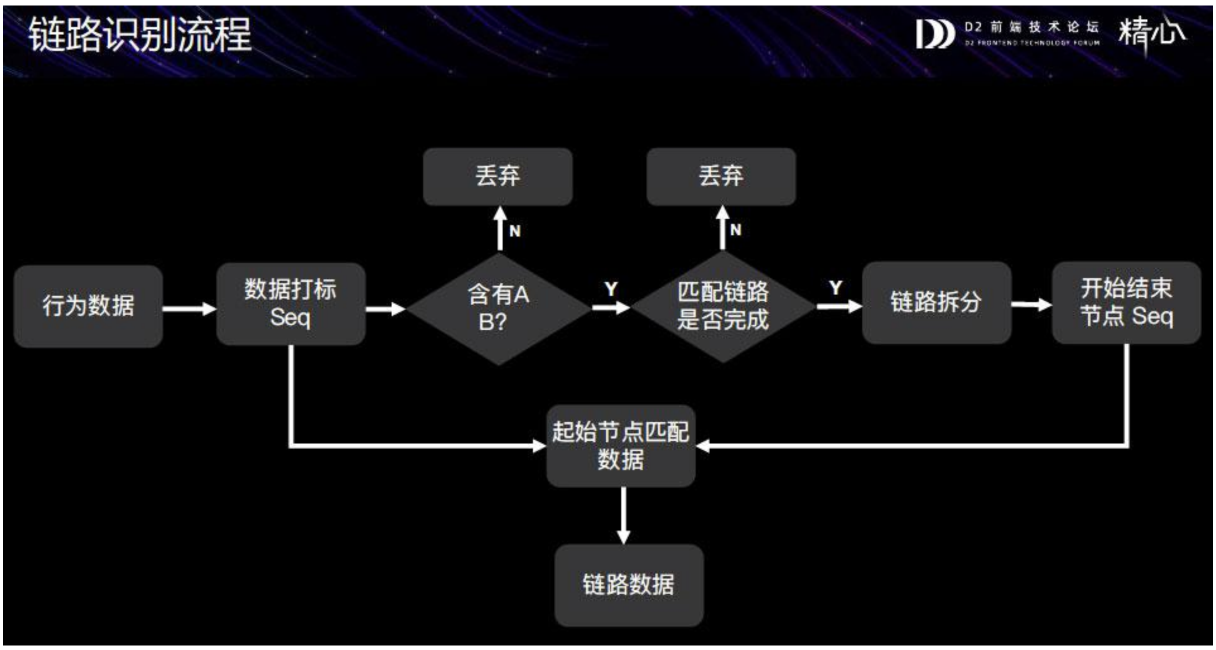
- 首先是数据采集，包括自动采集和主动上报；
- 第二个环节，是将数据进行一个清洗，自动对齐；
- 第三是链路识别，根据配置的起点和终点，将用户的操作力度识别出来；
- 链路识别后分为两个支路：
  - 第一个支路是数据挖掘，我们会去挖掘出节点的热度，固定的流量，最终将得到这个信息可视化维护；
  - 第二个支路是将用户的操作链路进行行为诊断，分析出链路中一些操作的问题，也汇总到这个信息，以及维护。

3）链路识别



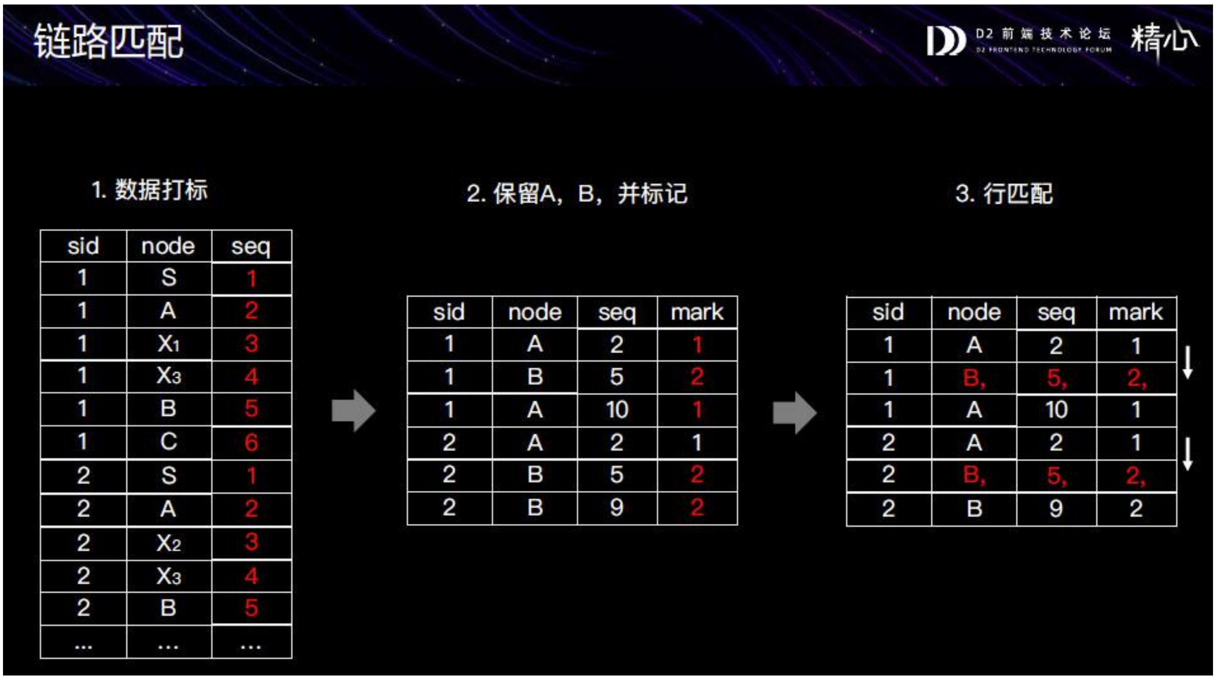
这是我模拟的两个用户在四次会话中的行为链路，我们现在以 A 为起始点，以 B 为终止点，来找到所有满足的这个用户行为链路。从这个简化的例子来看，我们很快就得到上面的这个用户链路图。用户从起点 A 分别到达 X1 X2 汇总在 X3 最终到达终点 B。

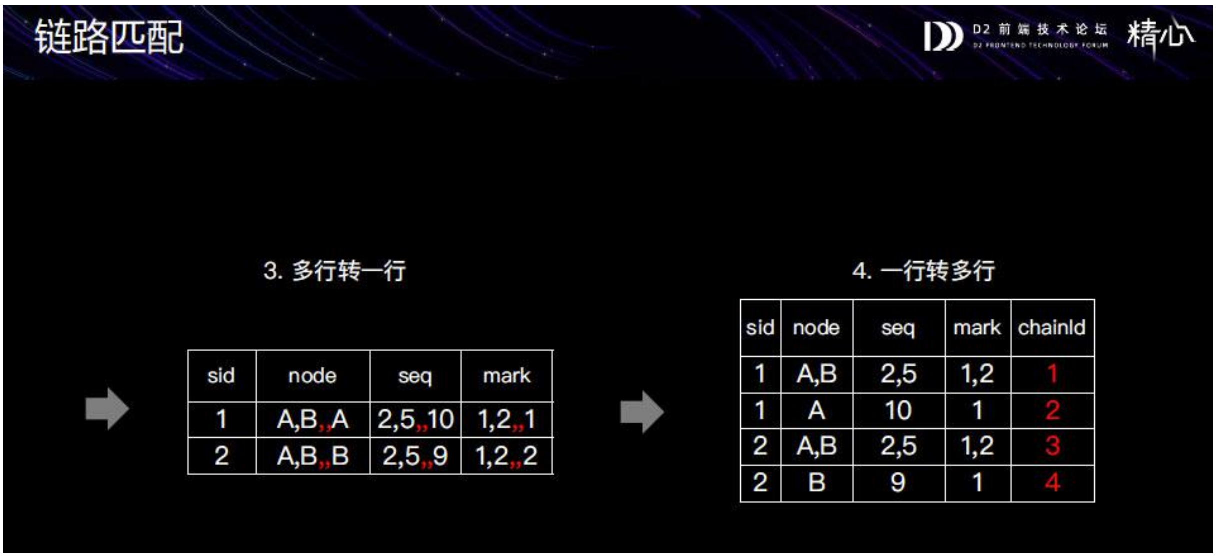
4）链路识别流程



这是简化后的链路识别流程。首先我们对行为数据打标，接着我们找到既含有 A 又含有 B 的用户，进行一个链路的匹配。看是否从 A 到 B 完成，当然一个用户也可能进行了多次，我们就要进去考虑链路拆分的问题，最后得到开始节点和结束节点的序号，然后和前面打好标的数据进行匹配，将 A 点和 B 点从所有的数据拿出来做聚合，就找到了这个数据。

5）链路匹配





我们在数据 ID 下打标，只保留 A 和 B，把 A 标记成 1，B 标记成 2，以时间顺序，从上到下来进行一个匹配，看是否 A 后面匹配到 B，如果匹配到，就在结束节点打一个逗号的标记，此时再做一个多行转一行操作，这样我们就可以发现，如果两个完整的链路，会被两个逗号分隔开，这样我们就再以两逗号去一行转多行，此时就把链路分割开来了，并且分配了 ID。这是一个简化后的识别步骤，实际情况会更复杂。

6）难点

数据产出慢：每天数十亿的海量数据，十几个步骤，识别算法复杂；
链路过于复杂：行为链路复杂，动作不可预知。

7）解决方案

性能解决方案



之前大部分的算法还有步骤，都耗在构造用户链路树上，我们考虑在数据层增加数据链路关系，将批处理转流处理的形式，优化中央处理节点，快速构造用户链路树，从而优化计算整体的耗时，达到一个实时计算的能力。

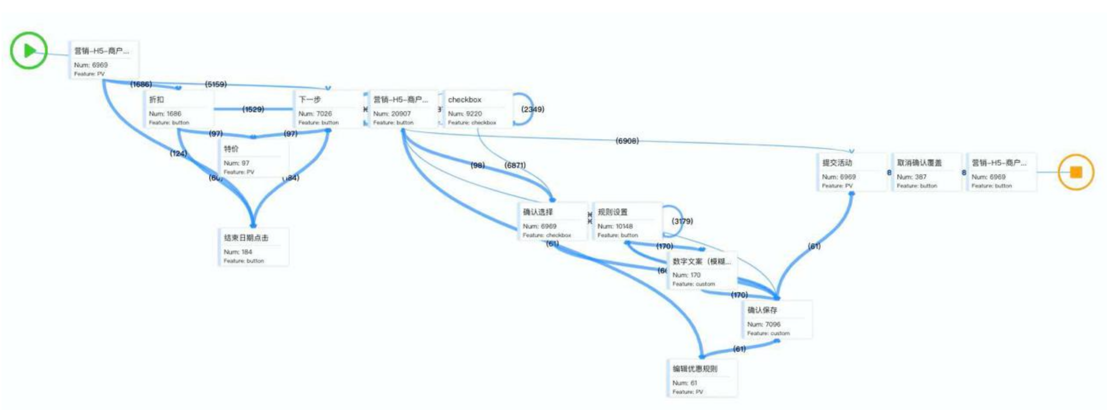
链路分散解决方案



当节点的文案有一些个性化话题的时候，或者链路中会有过多的节点的时候，会导致链路过于分散，失去趋势观测的意义。所以说我们提出了一个变动相似的概念，它包括节点相似和路径相似。节点相似就是从节点语义和 DOM 结构之间的维度来看，节点语义是 type/text/feature，DOM 结构是用 Xpath 去判断两节点是不是相似，对于路径相似，是看 K-邻接上下文。

2、链路可视化

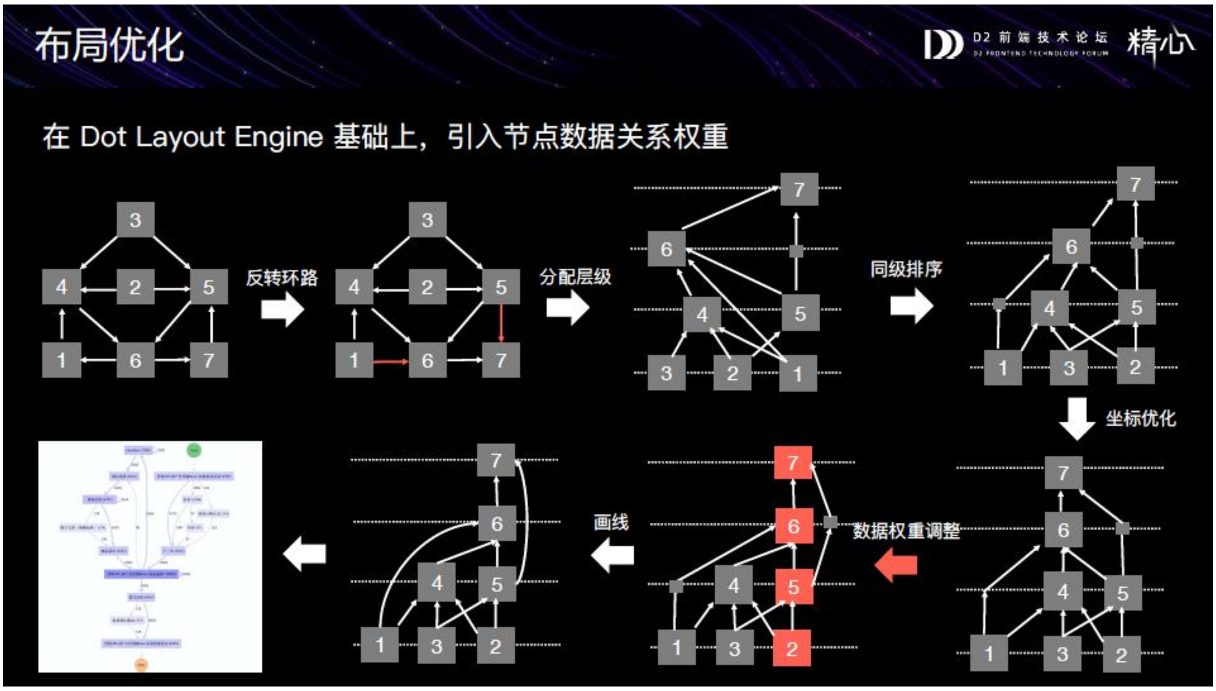
1）可视化尝试



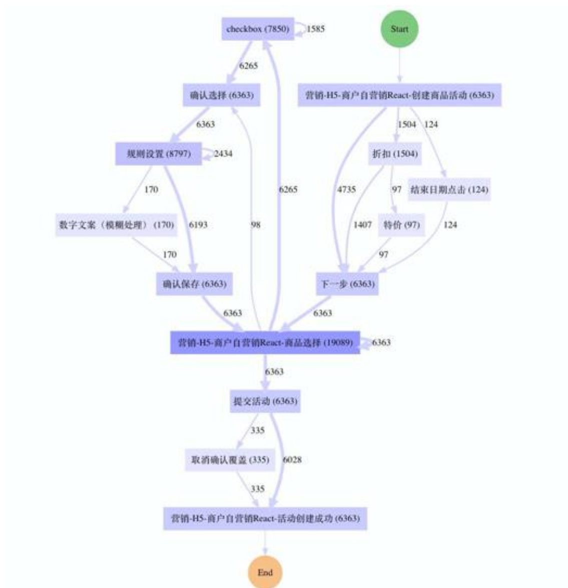
Top30 链路图（Dagre 布局）

Drage 布局是很多可视化库选择的方案，但是在用户操作链路的场景中，效果并不理想。

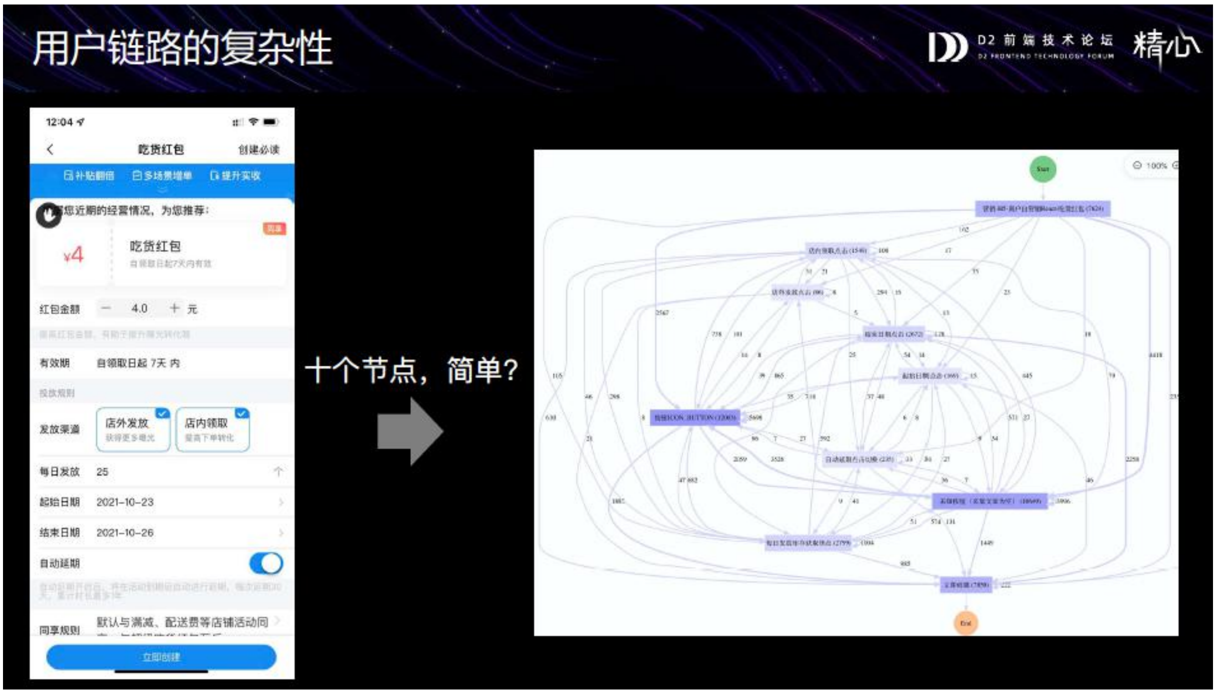
2）布局优化



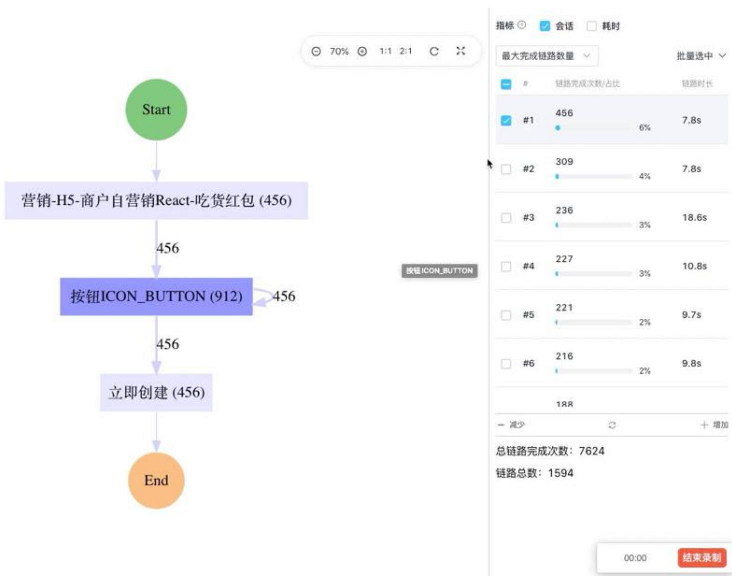
如图所示，是整个布局算法简化后的流程，可以看到在正常的坐标优化后，引入了数据权重，干预了节点布局，最后划线，得到了我们左边的记录可以看到，经过优化后的布局性能好了很多，整体的布局也比较合理，目前我们还在优化过程中。



3）用户链路的复杂性

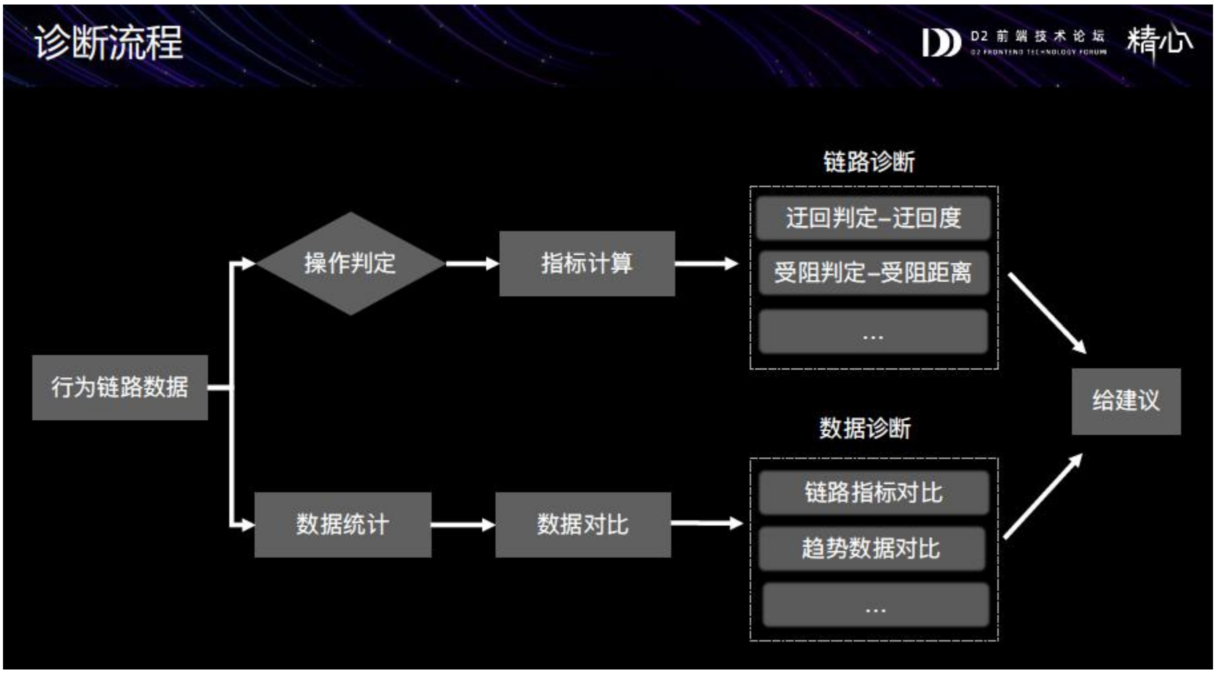


虽然经过了链路相似合并可视化的优化，但是在这个场景下，仅仅是十个节点就已经很复杂了，还是难以用肉眼或者说是简单的分析就能得出问题。从分析思想上进一步思考，也引入了流程的思想，将视角从全局的维度下沉，再经过多个拆分后，主要关注其热点 TOP 路径，并在用途内引入的颜色深浅来代表节点热度，最后得到下面的这个产品，这样就更易于我们人为去发现问题，发现产品的一些体验不好的地方。



3、链路诊断

1）诊断流程



链路诊断分为两部分：

- 首先是操作判定，识别出用户是属于哪一种异常操作，比如操作迂回，操作受阻等，判断结束后，会进一步的指标计算得出用户的迂回度、受阻距离等；
- 其次是数据统计，一定包括一个全量数据的统计，它的业务域下的数据工具，进行数据对比，来得出一些链路指标，区分和历史数据对比以及趋势的对比，基于这部分能够给出当前用户内路存在一些问题，最后给予一些改进建议。

2）操作判定

a.事件流匹配

- 复杂事件处理 CEP（Complex event processing)；
- 基于事件流的模式匹配(Pattern matching over event streams)；
- 应用：数据处理(Fink cep，Esper 等)，端智能等场景。



我们看一下场景中的一个处理流程。首先，将用户链路数据转化成事件流的形式输入到 JS 实现的这个 CEP 模式匹配器中，同时将我们实现的这个行为模式列表传送进去，最终输出我们的匹配结果。

定义模式

通过定义时间序列的模式来描述事件间的衔接关系，有 next、notNext、 followedBy、followedByAny、notfollowedBy。

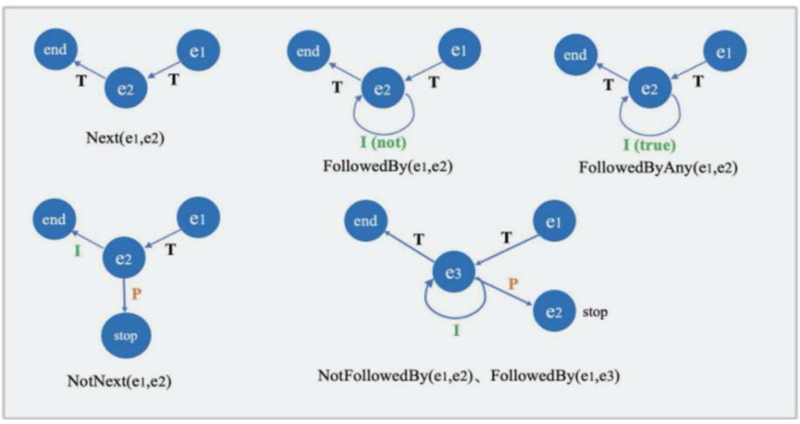
预设行为模式

定义好之后可以用来描述场景，比如：循环场景、单线循环。

- 单节点循环，a 到 a，我们就可以描述成
- Pattern.begin(A).next(A);
- 双节点循环，a 到 b，我们就可以描述成 Pattern.begin(A).next(B).next(A);
- 双节点间隔循环，a 到 b，就可以描述成 Pattern.begin(A).followedBy(B).followedBy(A).

其它场景依次类推，支路判定，操作受阻判定等场景进行规则描述。

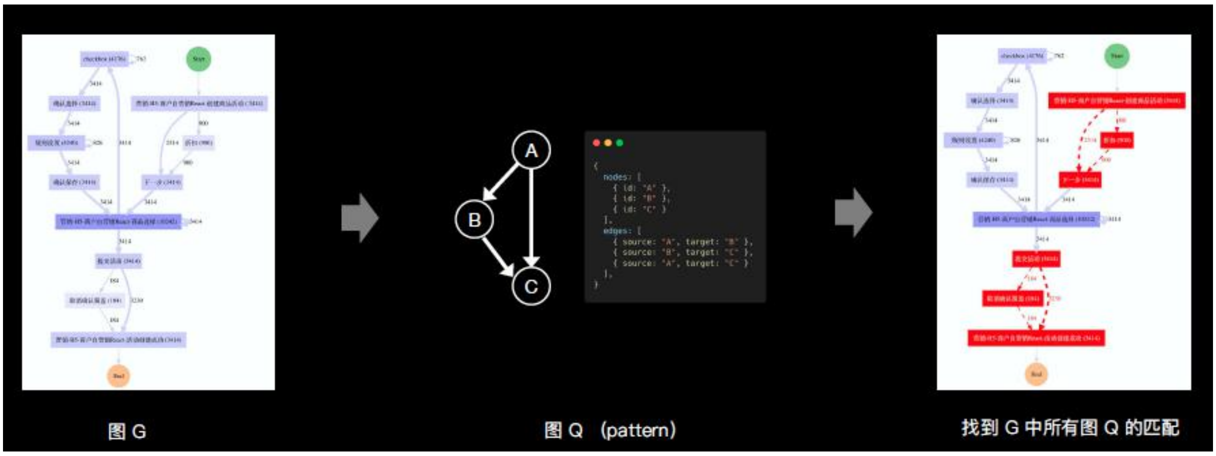
运行匹配



NFA（非确定的有限状态机）

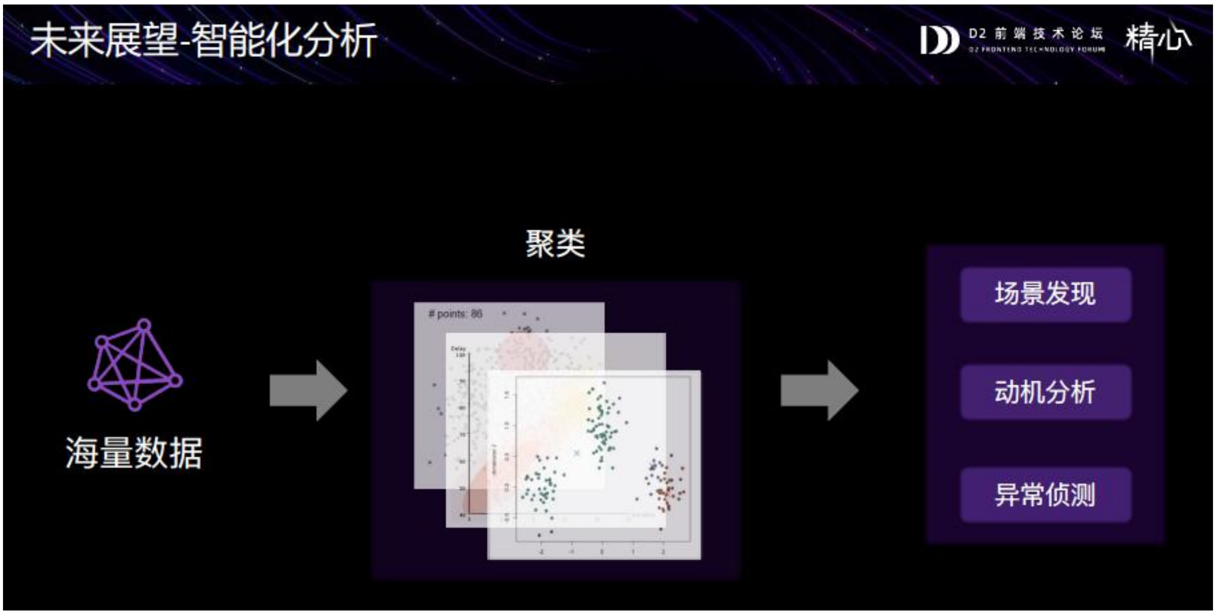
经过之前描述，我们会在下一步定义成一个 NFA（非确定的有限状态机），当没有事件触发的时候，状态机处于 Stop 节点，当有事件触发的时候，就会进行事件和规则的匹配，来决定是否前进、忽略、废弃，假设最终走到结束时，就表示这个匹配成功了；如果走到 Stop，或者说走到一个忽略，那就可以放弃已匹配的序列模式，最终直到符合这个描述的行为数据。

b.图模式匹配



图匹配（Graph Matching）是图同构（Graph Isomorphism）的进一步求解，复杂度未知，粗略估计属于 NP Hard，那我们这个场景，操作判定场景就是属于有向图的多次图匹配问题。我们有一张大图 G，同时又定义了一个图 Q（pattern），举个例子，就是 A 到 B 到 C，同时 A 又到了 C，而这个 pattern，可以描述成跟大图一样的结构，然后进行静态图的图匹配，最终在右侧的效果就是在大图中找到了这个图内所有的跟图 Q 属于同构的图。

4、未来展望



我们的核心目标是将已有的用户行为，更有价值的挖掘。我们在探索结合机器学习的聚类情况算法，做一些场景发现、动机分析以及异常行为监测。

- 场景发现，就是利用我们已有的链路数据，通过聚类算法，来找到关联业务场景的因子，来定义行为场景，这样定义了很细的场景，我们还能更精细化的来衡量不同业务场景下的体验。
- 动机分析，就是在已有的用户条件下，进一步的探究用户能够操作动机是什么，进而能够动态的给予用户一些干预和触达。
- 异常监测，是当我们运用聚类情况算法，把不同用户的行为捞出来之后，就可以进一步判定，用户是否为恶意用户。



适老化&无障碍的前端体验探索 — 为更多人提供便利和自由

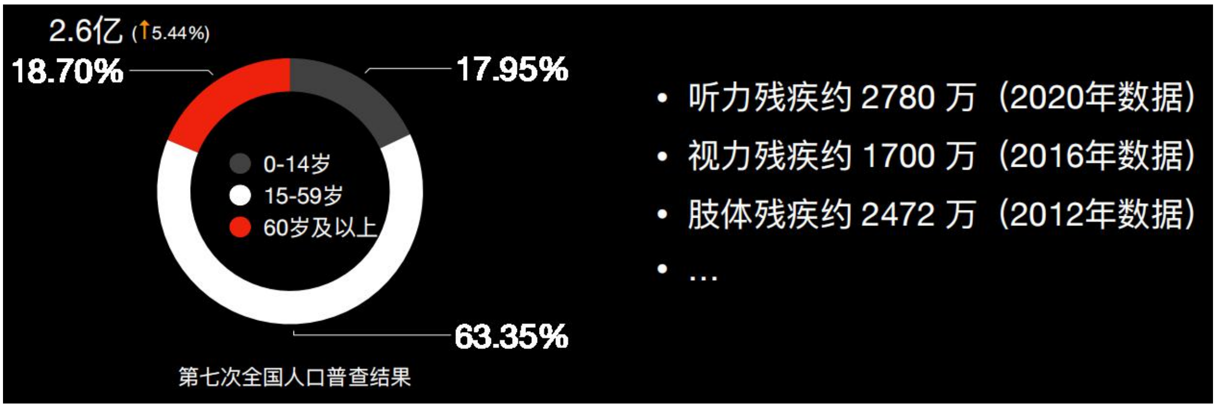
摘要：老年人和残疾人等群体由于身体机能缺陷或退化、信息技能水平不高、互联网应用适配不足等问题，在信息社会中逐渐处于弱势地位，难以享受技术进步带来的便利，面临使用型数字沟通问题。如何让所有人能平等、方便、无障碍地获取和利用信息，我们尝试从青松助手的产品设计，以及对腾讯问卷、腾讯兔小巢适老化及无障碍改造的经验中寻找答案。

分享人：陈洁婷，腾讯用户研究与体验设计部（CDC）高级前端开发工程师，先后参与过腾讯充值、米大师、腾讯设计云、腾讯用研云等项目，目前还主要负责腾讯问卷、腾讯兔小巢等用户研究生态项目“适老化&无障碍”改造及建设工作。

一、背景介绍

1、人口老龄化加深，残障人士数量大

据我国第七次全国人口普查显示，60 岁以上人口占总人口的 18.7%，较第六次普查结果上升了 5.44%，同时，残障人士数量也很大。



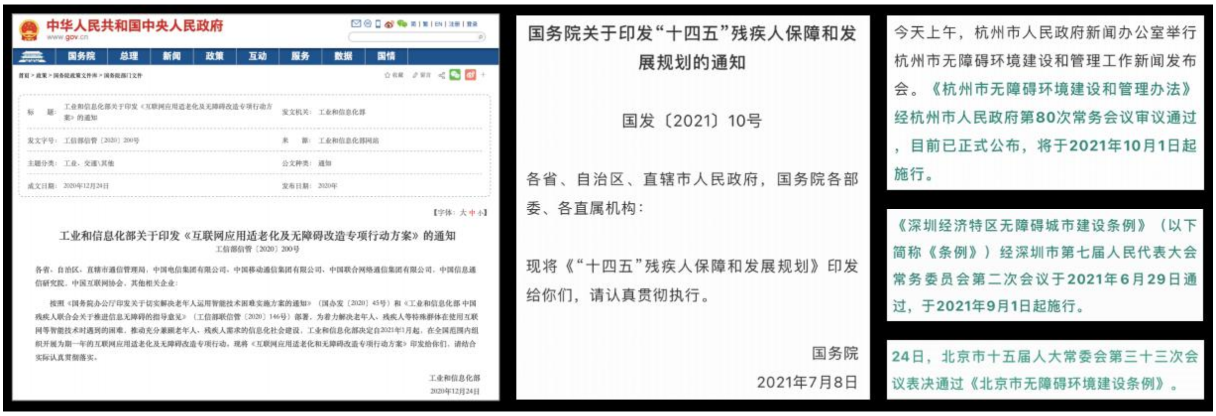
2、国内适老化、无障碍情况不容乐观





3、国家发出号召

工信部、国务院、各地政府印发通知、方案，奖惩结合（纳入企业信用体系）。适老化和无障碍改造相关内容，已经逐步从一种爱心公益行动发展为专项改造行为，并且有趋势在未来上升至法制法规层面。



中华人民共和国中央人民政府
www.gov.cn

国务院 | 总理 | 新闻 | 政策 | 互动 | 服务 | 数据 | 国情

首页 > 政策 > 国务院政策文件库 > 国务院部门文件

标　　题：	工业和信息化部关于印发《互联网应用适老化及无障碍改造专项行动方案》的通知	发文机关：	工业和信息化部
发文字号：	工信部信管〔2020〕200号	来　　源：	工业和信息化部网站
主题分类：	工业、交通\其他	公文种类：	通知
成文日期：	2020年12月24日	发布日期：	2020年

【字体：大 中 小】

工业和信息化部关于印发《互联网应用适老化及无障碍改造专项行动方案》的通知
工信部信管〔2020〕200号

各省、自治区、直辖市通信管理局，中国电信集团有限公司、中国移动通信集团有限公司、中国联合网络通信集团有限公司，中国信息通信研究院，中国互联网协会，其他相关企业：

按照《国务院办公厅印发关于切实解决老年人运用智能技术困难实施方案的通知》（国办发〔2020〕45号）和《工业和信息化部 中国残疾人联合会关于推进信息无障碍的指导意见》（工信部联信管〔2020〕146号）部署，为着力解决老年人、残疾人等特殊群体在使用互联网等智能技术时遇到的困难，推动充分兼顾老年人、残疾人需求的信息化社会建设，工业和信息化部决定自2021年1月起，在全国范围内组织开展为期一年的互联网应用适老化及无障碍改造专项行动。现将《互联网应用适老化和无障碍改造专项行动方案》印发给你们，请结合实际认真贯彻落实。

工业和信息化部
2020年12月24日

国务院关于印发“十四五”残疾人保障和发展规划的通知

国发〔2021〕10号

各省、自治区、直辖市人民政府，国务院各部委、各直属机构：

现将《“十四五”残疾人保障和发展规划》印发给你们，请认真贯彻执行。

国务院
2021年7月8日

今天上午，杭州市人民政府新闻办公室举行杭州市无障碍环境建设和管理工作新闻发布会。《杭州市无障碍环境建设和管理办法》经杭州市人民政府第80次常务会议审议通过，目前已正式公布，将于2021年10月1日起施行。

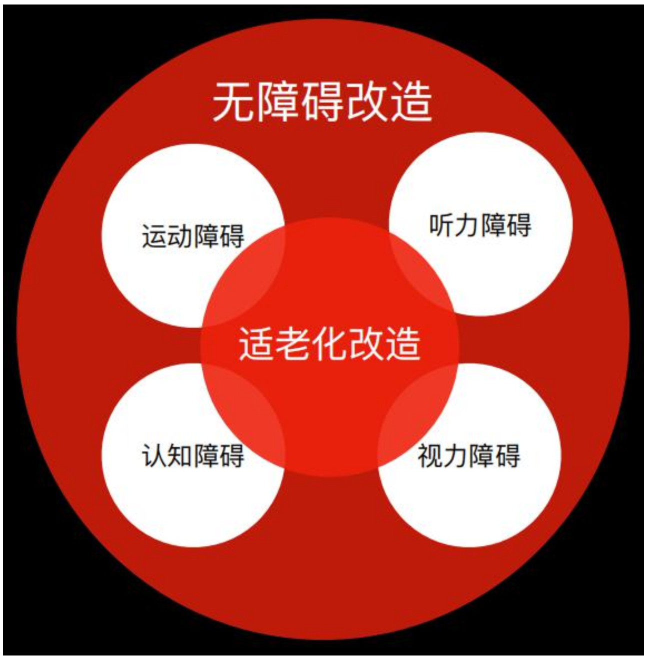
《深圳经济特区无障碍城市建设条例》（以下简称《条例》）经深圳市第七届人民代表大会常务委员会第二次会议于2021年6月29日通过，于2021年9月1日起施行。

24日，北京市十五届人大常委会第三十三次会议表决通过《北京市无障碍环境建设条例》。

二、改造思路

1、信息无障碍 Accessibility（a11y）

任何人（包括老年人和残障人）在任何情况下都能平等的、方便地、无障碍地获取信息、利用信息。



2、专项改造工作要顺利持续开展，需要多方协同

- 企业标准：指明改造范围、要求内容形成科学、可落实的标准；
- 实践方法：提供参考设计、评估流程，帮助了解初涉领域；
- 测评工具：支持快速自查自测，明确整改内容，提升效率；
- 样本库：招募样本，为各团队提供调研支撑；
- 优秀案例：知识积累，沉淀各类知识、案例、经验。

3、大处着眼：众人拾柴火焰高，发起、成立 Oteam

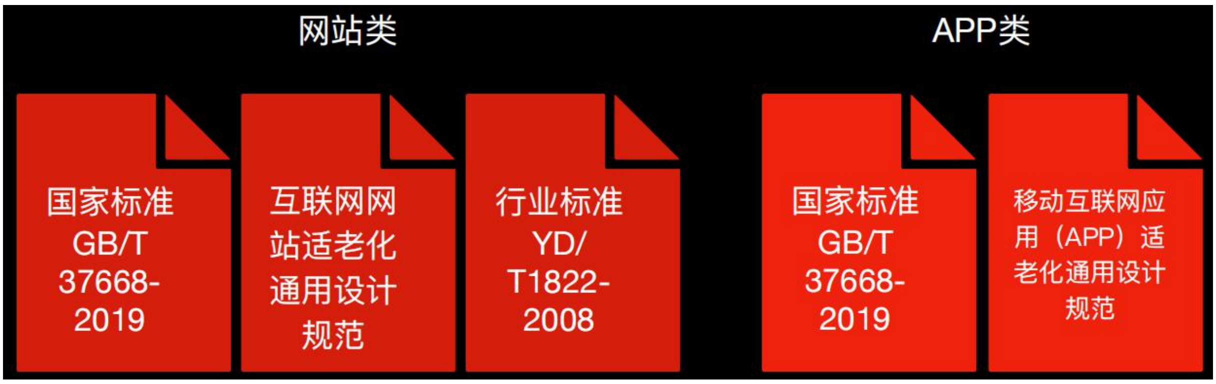
初期的协同规划，期望缔造全公司可持续的无障碍研发氛围，为普及社会无障碍服务提供助。Oteam 是腾讯内部一个开源协同的虚拟组织，参与者会共同完成从基础设施建设到落地产品开发的一系列工作。



三、具体方案

针对发现的问题，找到具体解决方案，目前发现的问题：

1、根据国际、行标文件



2、梳理“不适老”、“有障碍”的缺陷

根据国际、行标文件内容发现腾讯兔小巢存在的不适老和有障碍的缺陷。

- 不适老：
  - 字型大小无法调整；

  - 对比度不安全达标；
  - 焦点热区不达标；

- 有障碍：
  - 部分元素角色错误；
  - 部分元素无标签；
  - 部分元素无焦点；
  - 焦点冗余、穿透；

具体方案是根据国际和国内信息无障碍通用原则基于上述问题来制定的，通用原则包括可感知、可操作、可理解、鲁棒性和安全性。

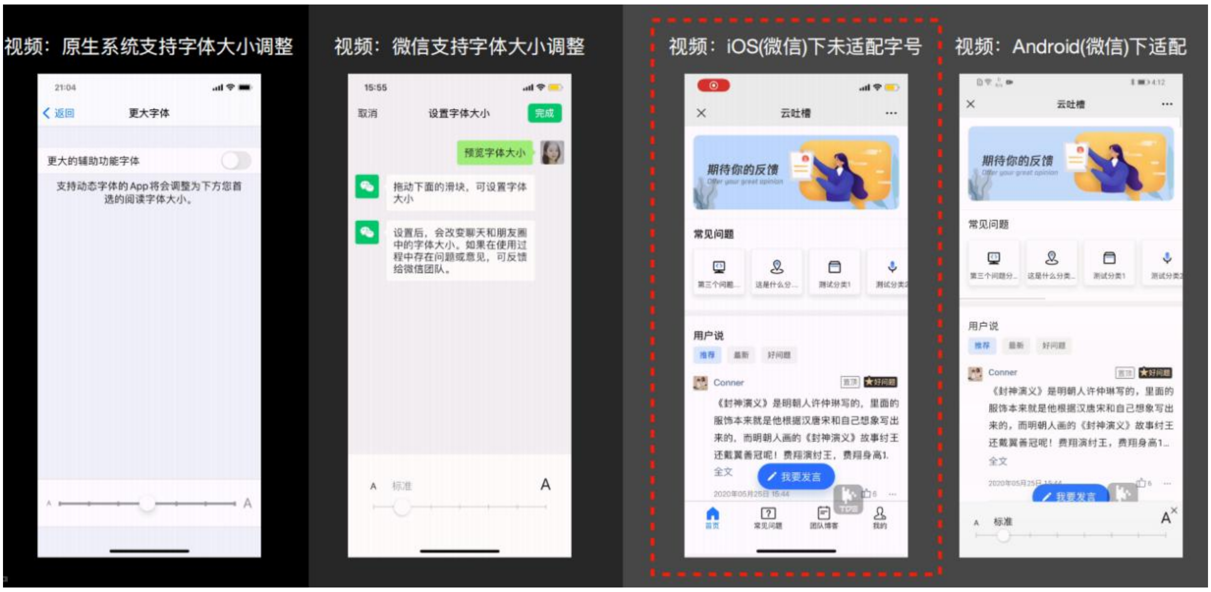
1）可感知

可感知包括：支持字型大小调整，关注信噪比，设计足够对比度，声明正确的角色，添加无障碍标签等。

a.大字体

问题：断裂的用户体验——设置字体大小失效。

最初的设计构想是使界面可以适配浏览器的字体大小，然而这一构想在 iOS 微信中失效。



通过定位问题发现错误是由于下面代码导致的：

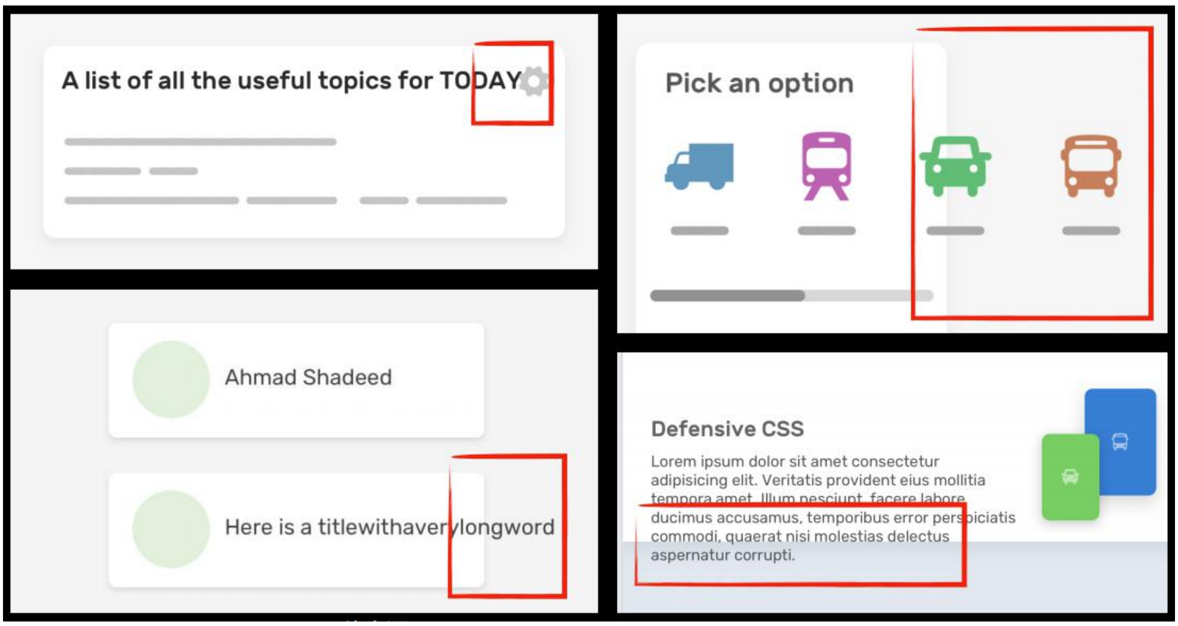
```
html {
  /* 字体大小不跟随 WebView 放大而放大 */
  -webkit-text-size-adjust: none;
}
```

另外，还要支持用户使用习惯比如双指缩放，横屏查看等：

```
<meta name="viewport" content="width=device-width, initial-scale=1.0,
  minimum-scale=1.0, maximum-scale=1.0, user-scalable=no">
```

思考：页面缩放就一键生效吗？

下图中问题的关键在于编写的 CSS 是否具有“防御性”。



改造后的效果：页面支持跟随 WebView 调整字体而缩放：



b.信噪比 - “大字体”带来的新问题

- 放大后的页面信噪比降低，需要适当调整交互。如下图改造中的页面字体放大后可读性降低，因此又进行了布局调整变成改造后的效果，页面的可读性和信噪比都有较高的提升。



- 次要内容可能抢占有限空间，需要精细化内容。



头像等比增大，压缩了有限的空间

次要信息（头像区）不跟随等比缩放

- 解决方案：

（1）合理使用不同长度单位，精细控制页面元素的缩放表现；

```css
.avatar {
  /* 不跟随字体大小变化 */
  width: 10vw;
  height: 10vw;
}

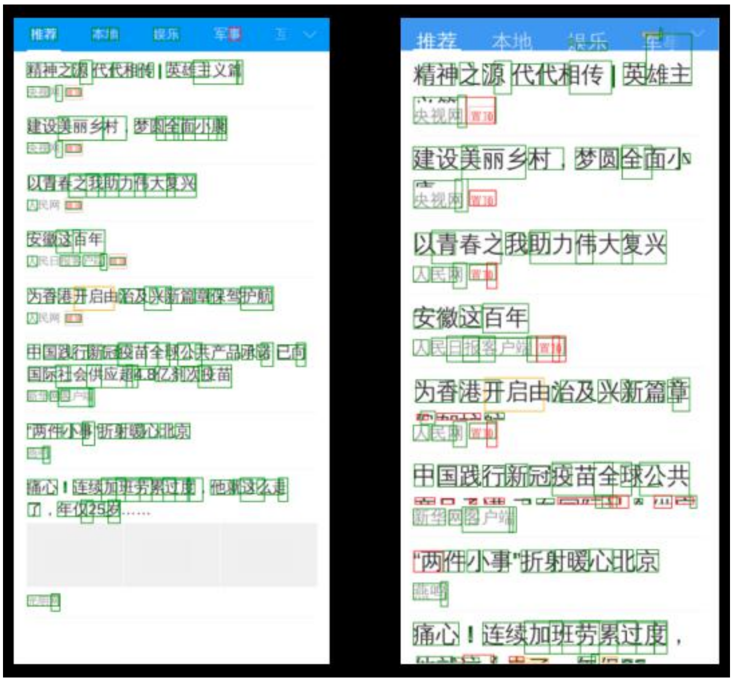
.card {
  /* 跟随字体大小变化 */
  width: 2rem;
  height: 2rem;
}
```

（2）自研的“大字号”检测工具：

第一步：通过脚本自动放大文字；

第二步：通过 ORC 分析缩放前后的页面截图；

第三步：根据文本识别情况计算页面适配评分。

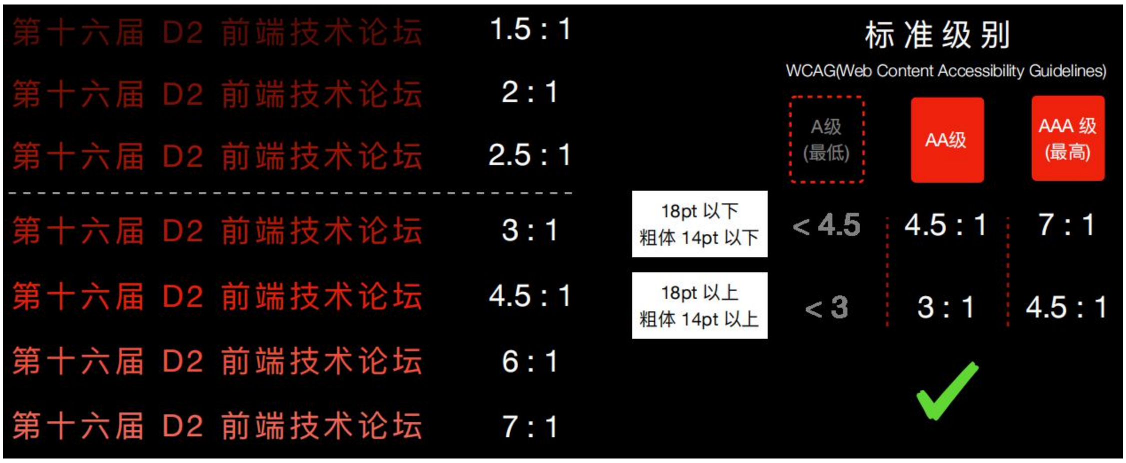


c.对比度

问题：“这颜色太浅了，这字我看不清楚”。

解决方案：使用不让眼睛“受累”的对比度。

下图左边一列的文字，自上而下的辨识度依次增大，国际 Web 无障碍指南针对于对比度的要求有一定的规范。



延伸：使用 prefers-*，提升页面的可访问性。

本质上是操作系统辅助功能的适配，也是体验设计连贯性的重要一环。

```css
body {
  /* 文字与背景对比度为 1.66*/
  background-color: #000;
  color: #333;
}

/* 提升对比度 */
@media (prefers-contrast: more) {
  body {
      /* 文字与背景对比度为 7.37 */
    background-color: #000;
    color: #999;
  }
}
```

```css
/* CSS媒体查询中新增的几个 prefers-* 特性功能 */

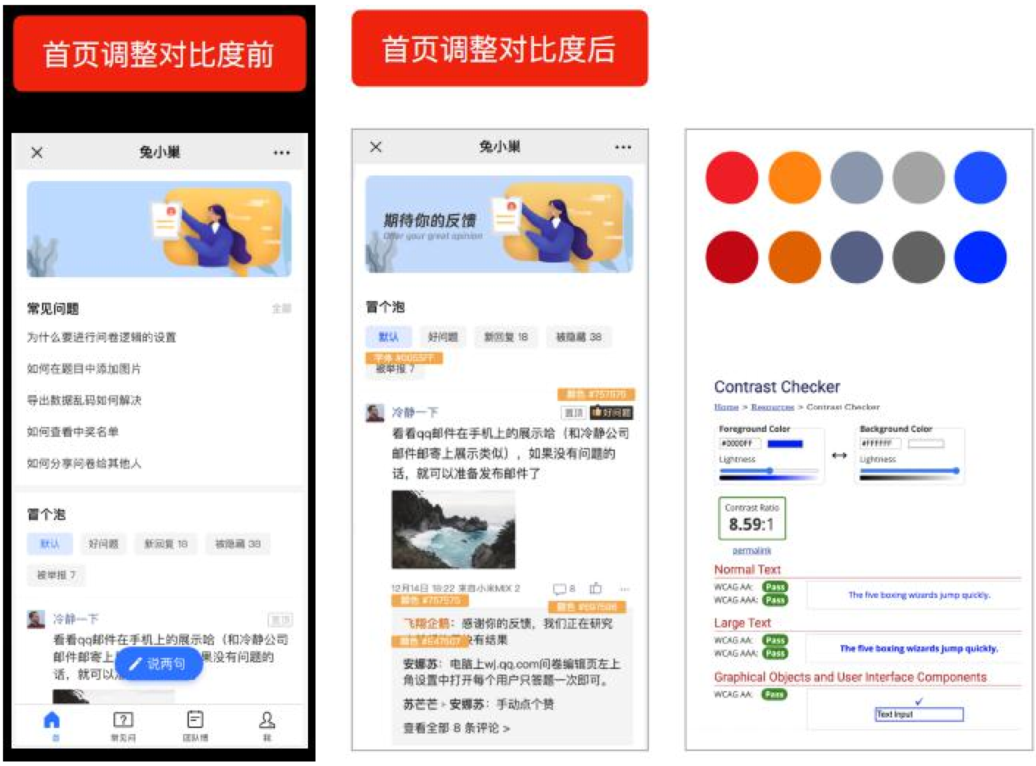
@media (prefers-reduced-motion) {
  /* 减弱动画效果 */
}

@media (prefers-color-scheme) {
  /* 适配明暗主题 */
}

@media (prefers-reduced-transparency) {
  /* 减少透明元素 */
}

@media (prefers-reduced-data) {
  /* 减少数据传输 */
}
```

调整对比度前后效果对比：



检测工具：应有尽有，都非常好用。



d.语义化

问题：图标按钮全都“听”不到。

- 视障人士使用读屏软件听到界面内容的主要操作有：
- 单指左右轻扫（选中元素）；
- 双指双击（激活元素）；
- 单指触摸（浏览、选中元素）；
- …

解决方案：语义化，辅以 WAI-ARIA 技术。

下图左边是改造前，空标签是读屏软件无法认读的内容，因此用户也就无从得知这些内容，图片则被认读成一串乱码让用户无法理解；改造后为每个元素定义了正确的角色，比如“按钮”、“链接”，图片则会有替代文本。



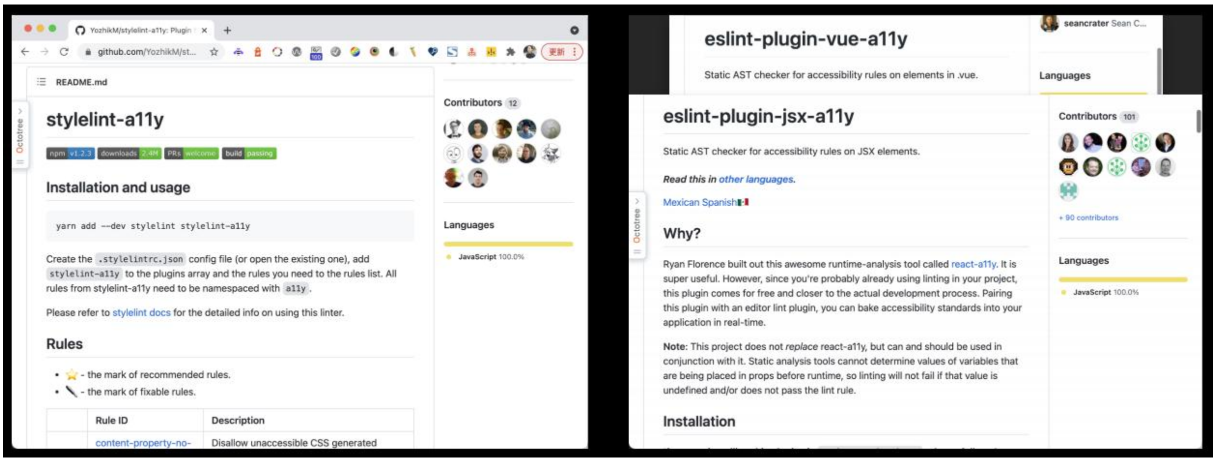
关键点：是使用正确的语义化标签(推荐)、WAI-ARIA 技术，让非装饰性元素能被辅助设备识别和操作。

以返回主页的图标为例，使用语义化标签的方式如下图：



检测工具：a11y 代码规范。

React 版共计 30+规则，能检测出大部分常见问题，Vue、Angular 也有配套的包。



2）可操作

可操作性包括：设置恰当的热区大小，设计恰当的手势操作等。

问题：可交互元素不容易被点到。

改造方案：44*44 point 苹果 HIG 建议的最小点触面积。



细节：具体情况具体分析。

例如为了避免误操作（“防错原则”），不同操作结果的按钮热区大小也会有差异。



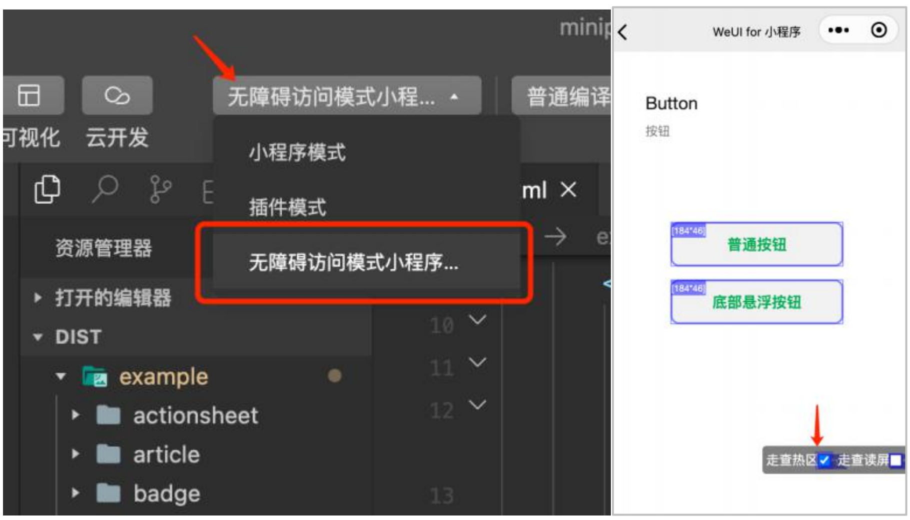
关键点：为所有可点击元素有针对性地定义大小合适的热区范围。

延伸：足够大的输出框等表单控件、点击热区位置设计（设置手指容易点到的位置）。

```
.icon-button {
  position: relative;
}
.icon-button::after {
    position: absolute;
    content: '';
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    min-width: 44px;
    min-height: 44px;
    width: 100%;  //可按需自定义
    height: 100%;  //可按需自定义
}
```

检测工具：微信小程序 - 无障碍访问模式插件。

路径：微信官方文档 - 小程序 - 开发 - 工具 - 无障碍访问。



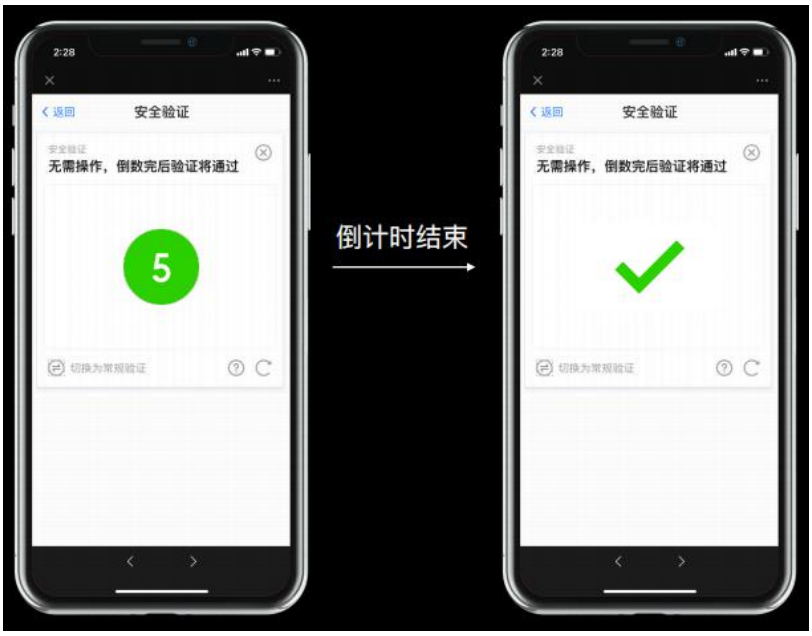
问题：过不来的验证环节。

老年人的肢体活动退化也导致了一些问题，比如在下图的验证环节，无法点到篮球、不理解拼图、不认识大小写字母等。



解决方案：通用的无障碍验证码。

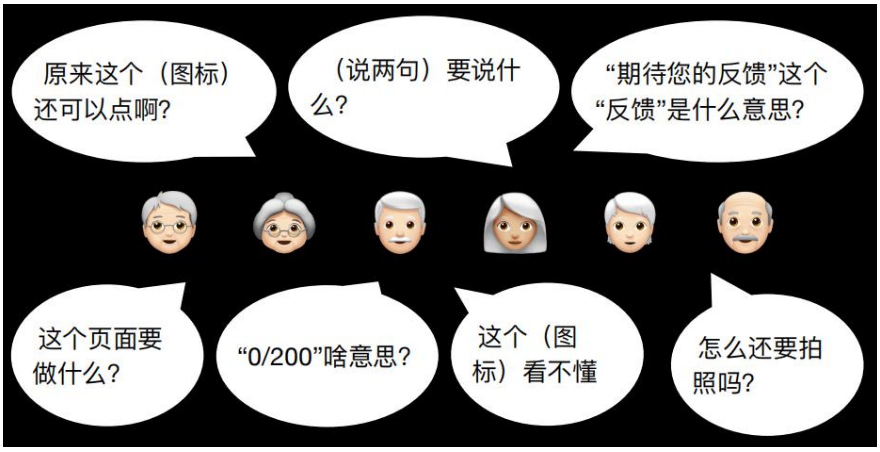
只需要“等一等”，验证就通过了，老年人、残障人士都能使用。



关键点：避免长按、拖拽/移、连续点击等对老年人不够友好的交互操作。

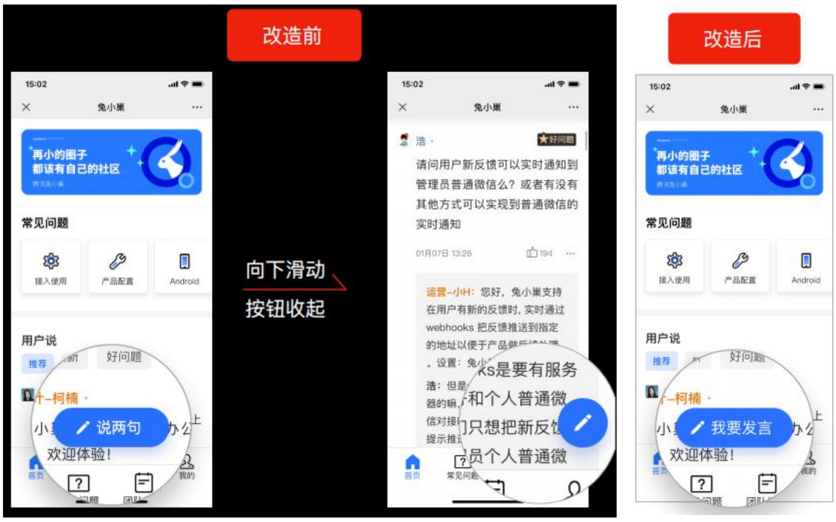
延伸：确保交互行为的简易性，例如上下滑动优于左右滑动，点击操作优于输入操作，支持一键操作等。

解决了不“适老”的问题后，我们邀请了伍大爷、周阿姨、吴大爷、翠姐姐、陈阿姨、项大爷、吴阿姨、许阿姨等叔叔阿姨们做了可用性测试，他们依旧遇到很多困惑（如下图），其核心问题是认知理解，因此，需要让长辈们能看到更能看懂。



3）可理解

问题：字和图标都看不懂，动画也不理解。



上图示例中改造前的问题：

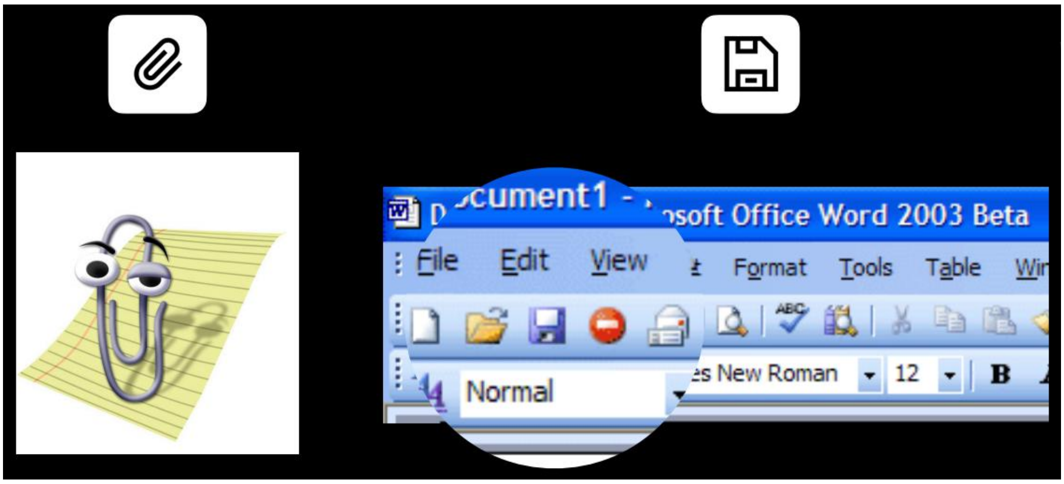
- “说两句”不理解，是评论？是用语音的方式说话？
- 展开时是用说话的方式输入，收起时是用写字的方式输入？
- 容易忽视“说两句”的图标及收起来的方式；

改造方案：

- 调整文案为更好理解的“我要发言”；
- 延长文字按钮的常驻时长；
- 向下滚动三屏内容以后再收起按钮。

下图展示了同龄人和长辈对图标的不同理解。由于没有经历图标变迁的过程，长辈们无法理解很多图标的含义。





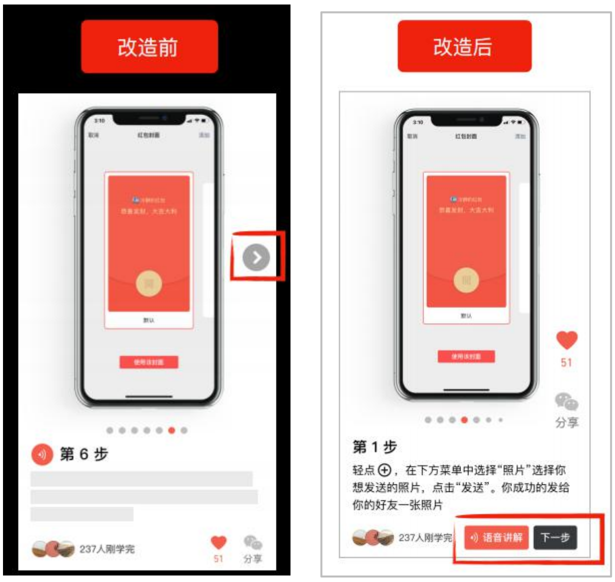
解决方案 1：使用更具象的图标。

越是具象的产品功能图标，越是容易被受试者理解。



解决方案 2：直接使用文字指引。

下图示例中改造前的“下一页”按钮在视频结束才出现，很容易被忽略，而且很多老人不理解图标的意思，因此，将图标直接改成语音和文字形式。



解决方案 3：给图标按钮添加文字描述（PC 端）。

下图示例来自 Google 翻译，鼠标悬停在图标按钮上的 popover 提示。



细节：“按下”效果是可交互元素不可缺少的一部分。

“秒回”的按钮更有安全感，由于有些按钮点击没有反应，让人误以为没有点上就会用力反复点。

改造后所有可交换元素按下后都有样式反馈。



关键点：建议给所有可交互元素设计、定义所有可能用到的状态样式。

```
button { /* 默认状态 */ }

button:hover { /* 鼠标悬停态 */ }

button:focus { /* 获得焦点时状态 */ }

button:active { /* 激活状态 */ }

button:disabled { /* 禁用状态 */ }
```

以 button 为例

思考：视频和图片哪种交互更符合预期?



改造前的问题：

- 视频教程没法缩放，图和字都看不清；
- 视频无法控制，有人嫌太快，有人嫌太慢；
- 容易忽略附带的语音信息；

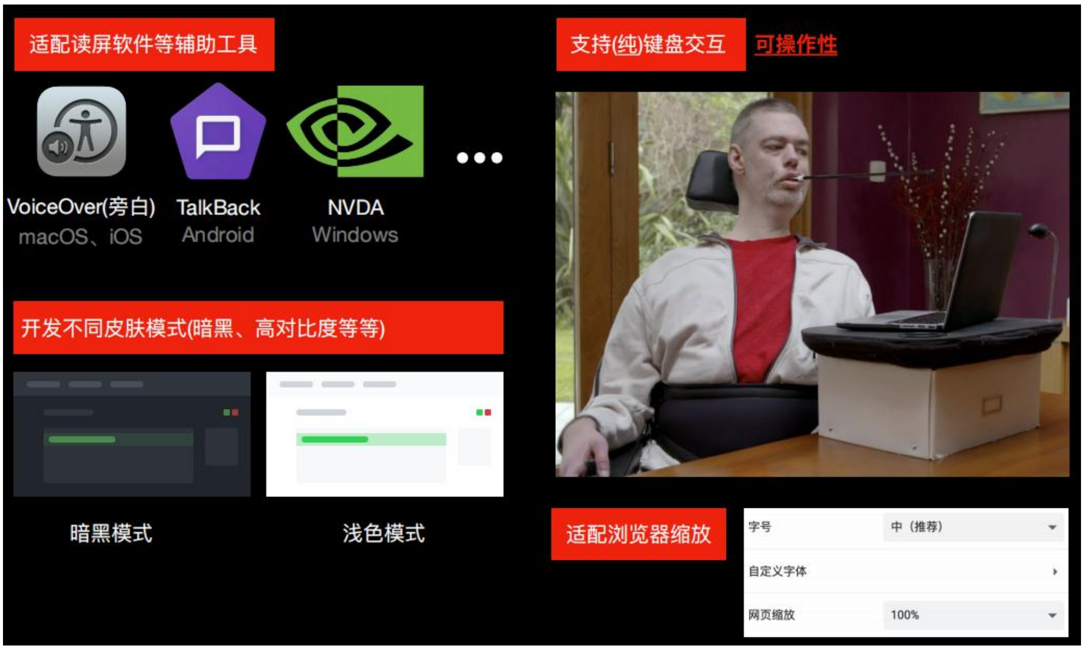
改造后，将视频改成图片：

- 图片支持双指拉划放大，解决了“看不清”的问题；
- 听着语音讲解依次翻页，觉得是在学习，在操作。

4）鲁棒性

是指与前端开发相关的辅助工具、功能。

辅助工具和功能包括（下图）：适配读屏软件等辅助工具，开发不同皮肤模式（暗黑、高对比度等等），支持（纯）键盘交互），适配浏览器缩放等。



5）安全性

安全性方面需要注意以下几点：

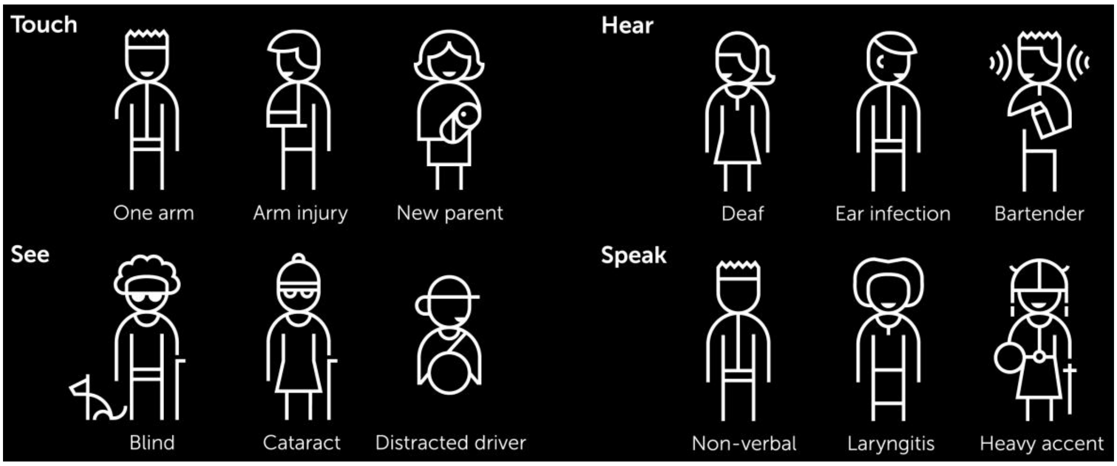
- 禁止广告插件；
- 禁止诱导类按键；
- 保障个人信息安全；

从上述无障碍改造示例中可以看到，老年人是各类障碍的轻度集合体，而每个人都会变老，因此改造工作彼此互惠。我们用一张图总结以上 5 点：



6）永久性障碍 Vs 情景式障碍

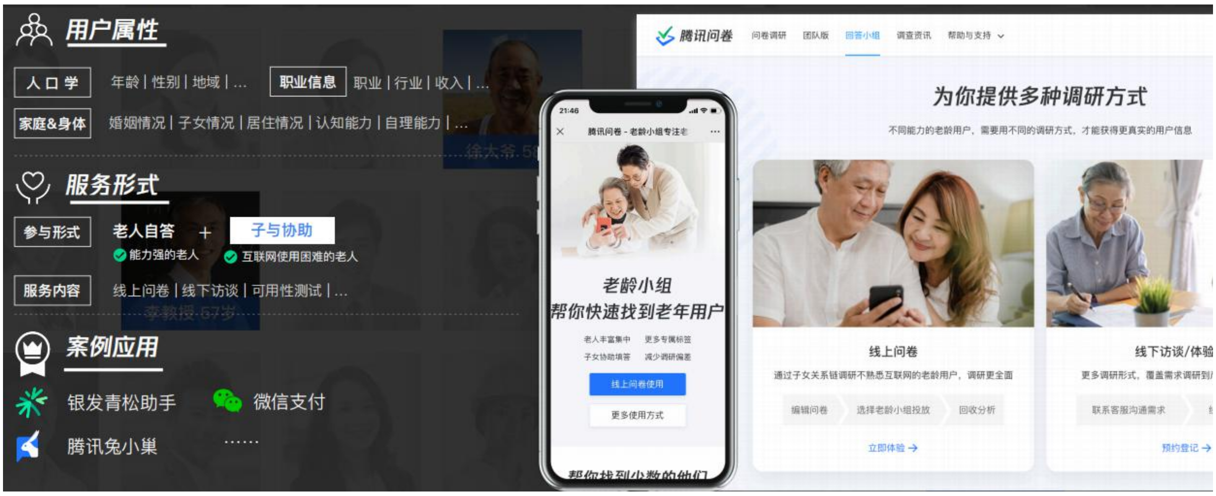
除了老年人和残障人士，普通人也能因适老化、无障碍改造而受益，比如：抱着孩子只能用一只手、摔伤手臂或腿、有口音的人，等等。



聚沙成塔，我们希望能做一本“信息无障碍”的百科全书，让更多的人了解信息无障碍，提升“适老化”、“无障碍”意识，了解“信息无障碍”需要做什么、怎么做。



腾讯问卷的老龄小组，针对老年人的问题调研可以供大家借鉴。老龄小组已经招募老龄样本 2.4 万，积累 14 类标签属性，已服务 14 个业务调研，回收 511 个老人意见，助力提升产品适老化体验。



参考网址：https://wj.qq.com/elderly_group.html

四、总结展望

适老化和无障碍改造工作的主要目的是为提升用户体验，为更多人带来便利、自由和幸福感。最后给大家推荐几本书，在这方便如果跟技术相关我们要做什么的话，其实就是做好最基础的语意化，去写友好性的、防御性的 CSS 就可以了。



欢迎对适老化和无障碍改造有兴趣的朋友加入我们。

加入我们　　更多案例

低代码

腾讯低代码引擎建设思考和实战

摘要：LCDP 日渐火热的当下，各大企业都推出自己的低代码开发平台，甚至很多大企业内部都有多套低代码开发平台。在 LCDP 为企业开发降本增效的基础上，如何将低代码底层能力进行规范，进一步挖掘低代码技术的深水区，统一并丰富整个低代码研发生态，显得尤为重要。本文将结合腾讯低代码开发平台 Oteam 和内部多个 LCDP 产品案例的具体实践，对低代码底层 UI、逻辑、DSL、生产运行、质量保证等多个引擎模块进行阐述。

分享人：丁涛(MarsboyDing)，腾讯互娱高级前端工程师，腾讯星图低代码平台前端负责人。

目录：

一、通用能力架构

二、通用能力引擎设计

三、内部低代码平台的实践

本篇内容将通过通用能力架构、通用能力引擎设计、内部低代码平台的实践来介绍腾讯低代码引擎建设的思考和实战。

一、低代码通用能力架构方案

1、概念

如何定义低代码?

- 广义：所有可以降低生产成本，减少代码开发的方式、过程或者工具/系统；

- 狭义：指通过图形化、可视化的开发方式，基于模型驱动和逻辑编排的能力,进行拖拽和配置，最后生成我们想要的产物(应用程序、接口等)。

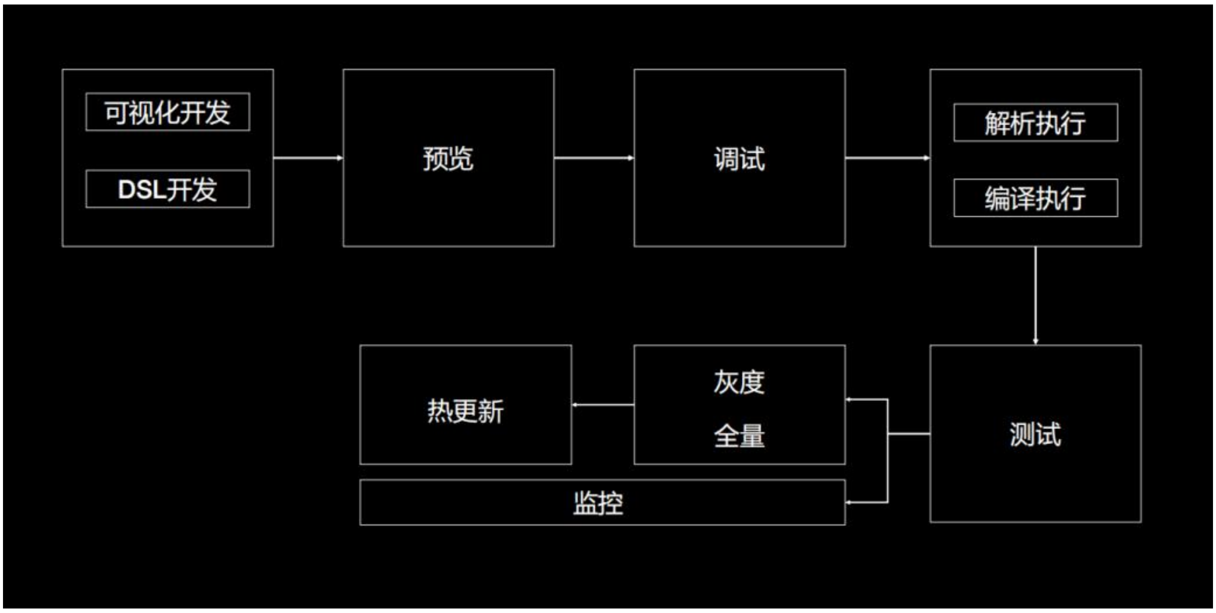
2、低代码、无代码、纯代码

- 无代码：不需要编辑任何代码的被称为无代码也可称为 0 代码；
- 纯代码：手写代码，包括各种实现；
- 低代码：在（0，100）的量化空间内，低代码的定位很灵活，不同的量化定位，决定了低代码平台的应用形式和特征。

无代码与纯代码的区别：

	开发灵活性	**人员友好性**	**运行稳定性**	**封装性**	**复杂需求**
无代码	低	容易上手	好	好	难以实现
纯代码	高	专业性高	差	差	可实现

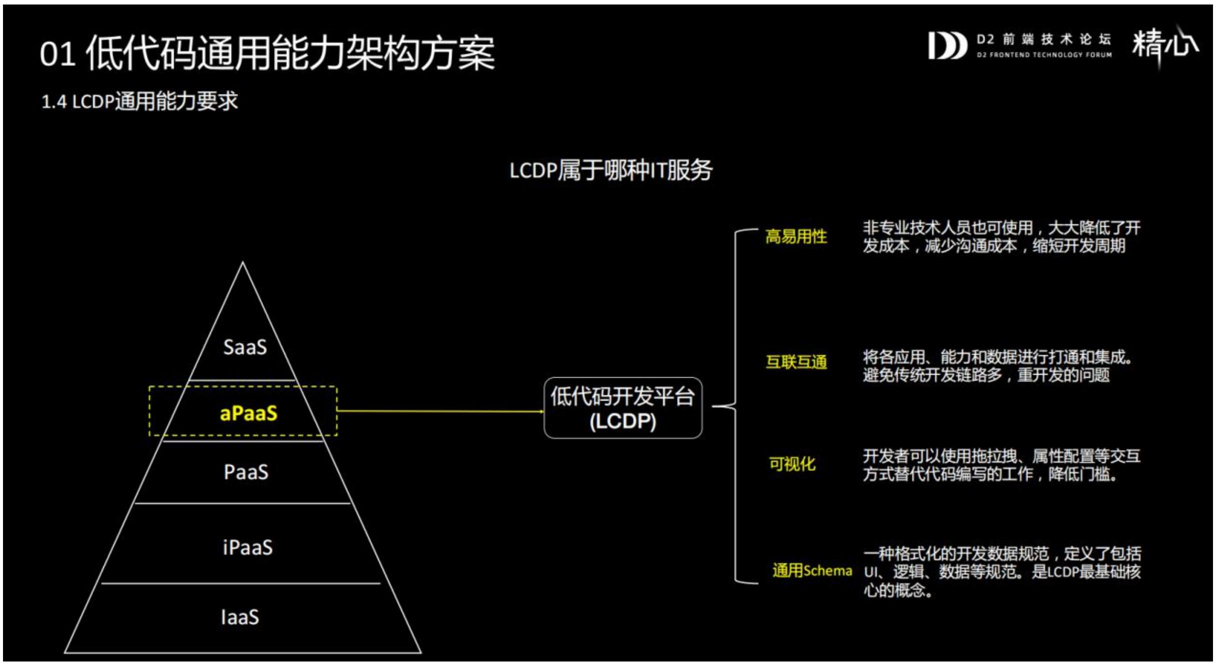
3、全生命周期的 LCDP



全生命周期的 LCDP 具体步骤：

可视化或 DSL 开发→预览→调试→生产运行（解析执行、编译执行）→测试→灰度或全量发布（在发布的同时监控也会起作用）→热更新（程序有一些改动、变更可以通过热更新来进行更新）

4、LCDP 通用能力要求



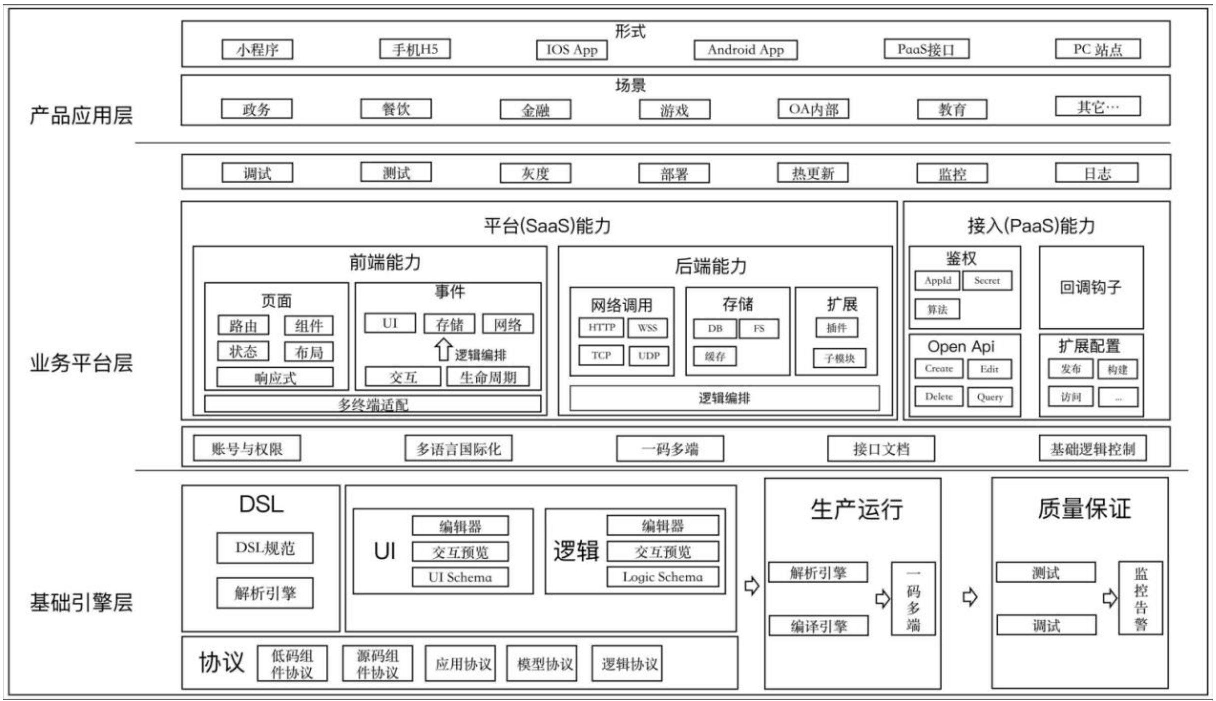
aPaaS 是什么？

aPaaS 就是低代码开发平台所处的位置，它有以下四个特点：

- 高易用性：非专业技术人员也可使用，大大降低了开发成本，减少沟通成本，缩短开发周期；
- 互联互通：将各应用、能力和数据进行打通和集成。避免传统开发链路多，重开发的问题；
- 可视化：开发者可以使用拖拉拽、属性配置等交互方式替代代码编写的工作，降低门槛；
- 通用 Schema：一种格式化的开发数据规范，定义了包括 UI、 逻辑、数据等规范，是 LCDP 最基础核心的概念。

aPaaS 具体的能力架构

(来自腾讯低代码企业规范)



- 基础引擎层：DSL、UI、逻辑、生产运行、质量保证；
- 业务平台层：开发平台（SaaS）能力、接入（PaaS）能力；
- 产业应用层：
    - 形式：小程序、IOS App、PC 站点等；
    - 场景：政务、餐饮、金融游戏等。

5、腾讯低代码 Oteam 的建设架构

Oteam 是什么

在腾讯，Oteam 是公司级的一种开源协同项目组织形式。主要是为了统-公司内某一领域相关项目的建设,减少重复轮子的建设,由各相关 BG、部门同学一起参 与协同共建。

腾讯低代码 Oteam 的目标

在公司内，低代码 Oteam 旨在建设统一的低代码开发平台(LCDP)的通用底层引擎、SDK。为更多的低代码开发平台提供最底层统一能力接入 ,助力各领域，各场景下 LCDP 上层平台的建设。



- Oteam 基础组件：UI 可视化开发、逻辑可视化开发、代码语言、生产与运行、质量保证；
- 开发工具/系统：星图、Tapd、阿波罗、腾讯云微搭、无极；
- 应用/产品：Web/H5 营销活动、小程序、快速建站、管理系统、开发 App、后端接口。

二、通用能力引擎设计

通用能力引擎设计将从以下 5 个方面讲解：UI 可视化开发、逻辑可视化开发、代码语言、生产与运行、质量保证。

1、UI 可视化引擎



关键词

UI 可视化引擎关键词：拖拉拽、UI 组件、配置事件、预览、编辑样式、多页面。

长啥样



以微搭低代码和阿波罗低代码为例，从图中可以看出，是由物料区、拖拽预览区、表单配置区三部分组成。

如何定义和描述 UI 的内容

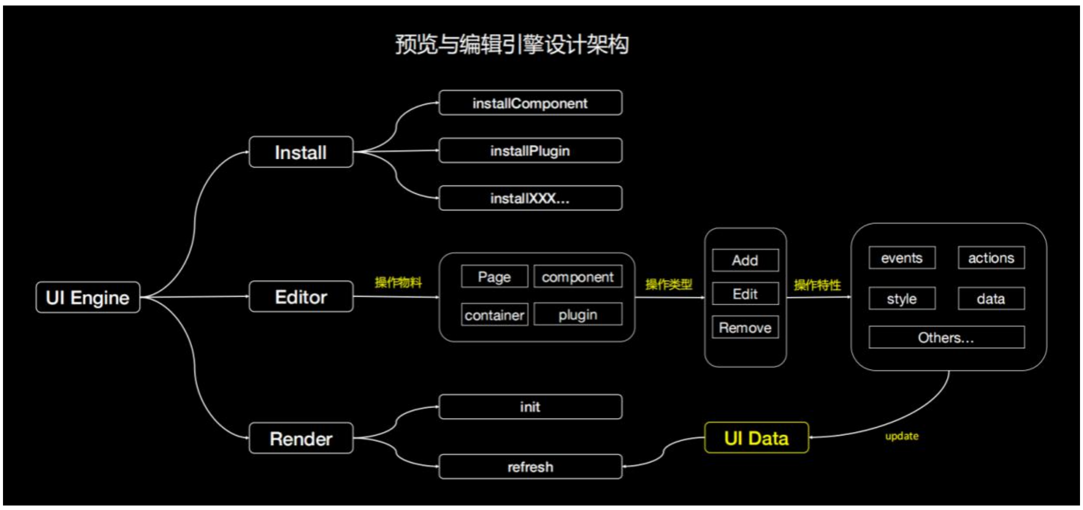


```
{
  "type": "page",
  "data": {
    "domain": "https://syn0.map.qq.com",
    "awardApiPath": "/api/award",
    "actInfoApiPath": "/api/actInfo",
    "actId": "10001"
  },
  "containers": [
    {
      "type": "container",
      "id": "container1", // 全局唯一的id，用于索引
      "name": "Custom-Activity01", // 自定义容器
      "layout": "static", // 默认为单列布局
      "data": {
        "title": "公交绿色出行活动",
        "shareInfo": {
          "title": "绿色出行",
          "description": "...",
          "link": "...",
          "imgUrl": "https://4gimg.map.qq.com/a.png"
        }
      },
      "components": [ // 常规组件
        {
          "type": "component",
          "id": "header1",
          "name": "Custom-Header",
          "data": {
            "title": "${title}",
            "statusBarHeight": 40,
            "bannerImgUrl": "...",
            "cash:react": 0 // 带:react标识意味这个变量发生变更会触发组件或子组件渲染，或者触发某些逻辑
          },
```

UI Schema 分为以下几部分：

- Schema Core:
  - 特性定义：数据、布局、样式、事件、生命周期、动作；
  - 物料定义：页面、容器、组件、插件；
- H5 Schema：定位布局、更多业务组件；
- Console Schema：流式布局、丰富表单组件；

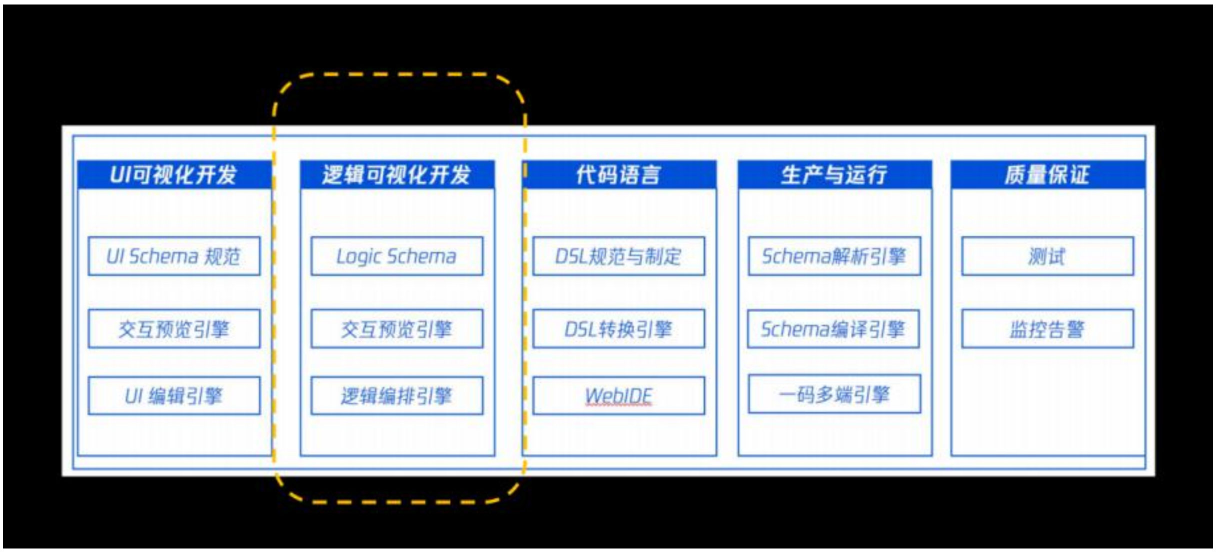
预览与编辑引擎设计架构



UI Engine:

- Install: installComponent、installPlugin、installXXX…
- Editor:
  - 操作物料：Page、component、container、plugin;
  - 操作类型：Add、Edit、Remove;
  - 操作特性：events、actions、style、data;
- Render: init、refresh;

2、逻辑可视化开发



关键词

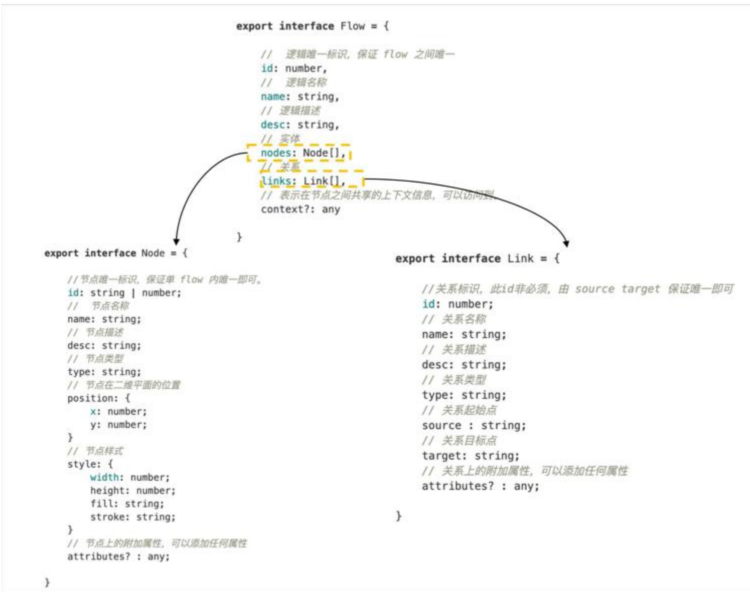
逻辑可视化引擎关键词：拖拉拽、流程连线、条件控制、预览、网络与 DB、微服务。

长啥样



以腾讯星图低代码为例，从图中可以看出是由：功能节点区、流程编排区、表单配置区三部分组成。

如何定义和描述逻辑的内容



```
export interface Flow = {

    //  逻辑唯一标识，保证 flow 之间唯一
    id: number,
    //  逻辑名称
    name: string,
    // 逻辑描述
    desc: string,
    // 实体
    nodes: Node[],
    // 关系
    links: Link[],
    // 表示在节点之间共享的上下文信息，可以访问到
    context?: any

}

export interface Node = {

    //节点唯一标识，保证单 flow 内唯一即可。
    id: string | number;
    //  节点名称
    name: string;
    // 节点描述
    desc: string;
    // 节点类型
    type: string;
    // 节点在二维平面的位置
    position: {
        x: number;
        y: number;
    }
    // 节点样式
    style: {
        width: number;
        height: number;
        fill: string;
        stroke: string;
    }
    // 节点上的附加属性，可以添加任何属性
    attributes? : any;

}

export interface Link = {

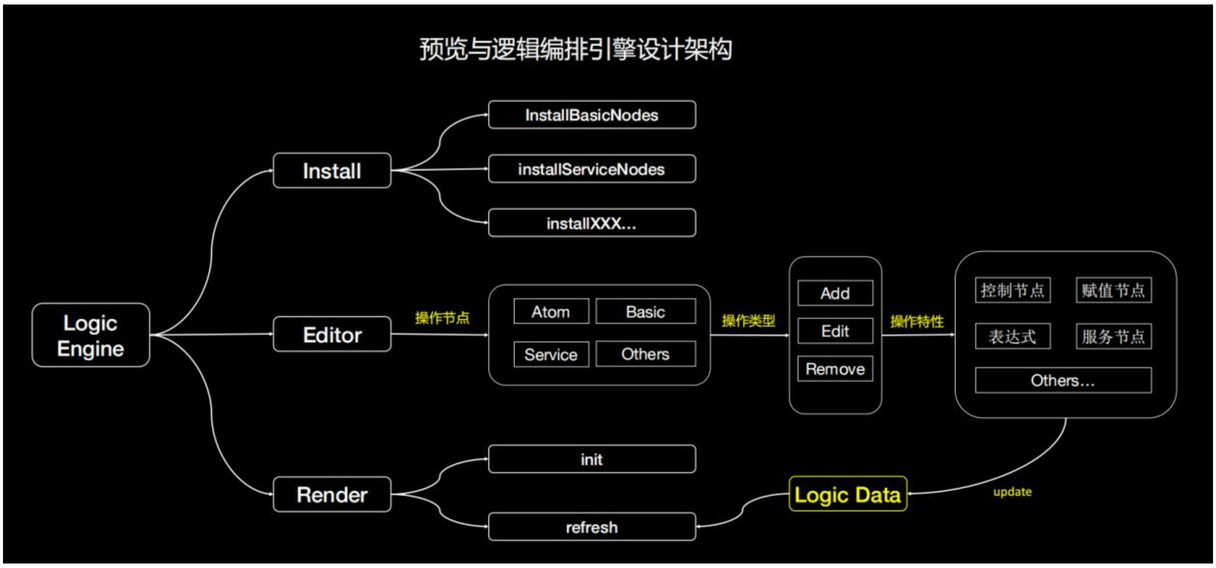
    //关系标识，此id非必须，由 source target 保证唯一即可
    id: number;
    // 关系名称
    name: string;
    // 关系描述
    desc: string;
    // 关系类型
    type: string;
    // 关系起始点
    source : string;
    // 关系目标点
    target: string;
    // 关系上的附加属性，可以添加任何属性
    attributes? : any;

}
```

Logic Schema 分为以下几部分：

- Schema Core:
  - 节点特性：判断、赋值、循环、网络调用、存储操作;
  - 节点类型：原子服务、流程控制、通用服务、扩展服务;
- Front Schema：UI 操作、页面操作;
- Backend Schema：云服务调用、数据库调用;

预览与逻辑编排引擎设计架构



Logic Engine:

- Install:installBasicNodes、installServiceNodes、installxxx…
- Editor:
  - 操作节点：Atom、Basic、Service、Others
  - 操作类型：Add、Edit、Remove
  - 操作特性：控制节点、赋值节点、表达式、服务节点
- Render:init、refresh

3、代码语言（DSL）



DSL(领域专用语言)

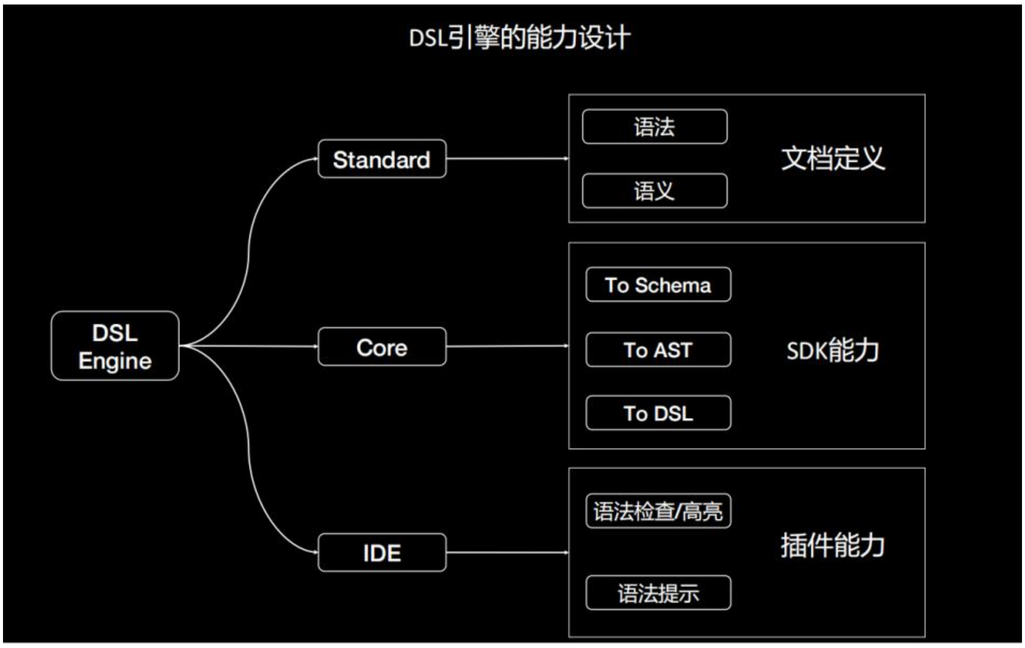
DSL 的特征与必要性：

- 特性：一种语言、特定领域、尽量简单、转换成传统代码
- 必要性：与可视化形成补充、开发人员友好、脱离 UI 编辑器、跨平台二次开发

DSL 在 LCDP 中的位置

可视化开发与 DSL 形成切换与补充，到 Schema,再到解析与编译。

DSL 引擎的能力设计



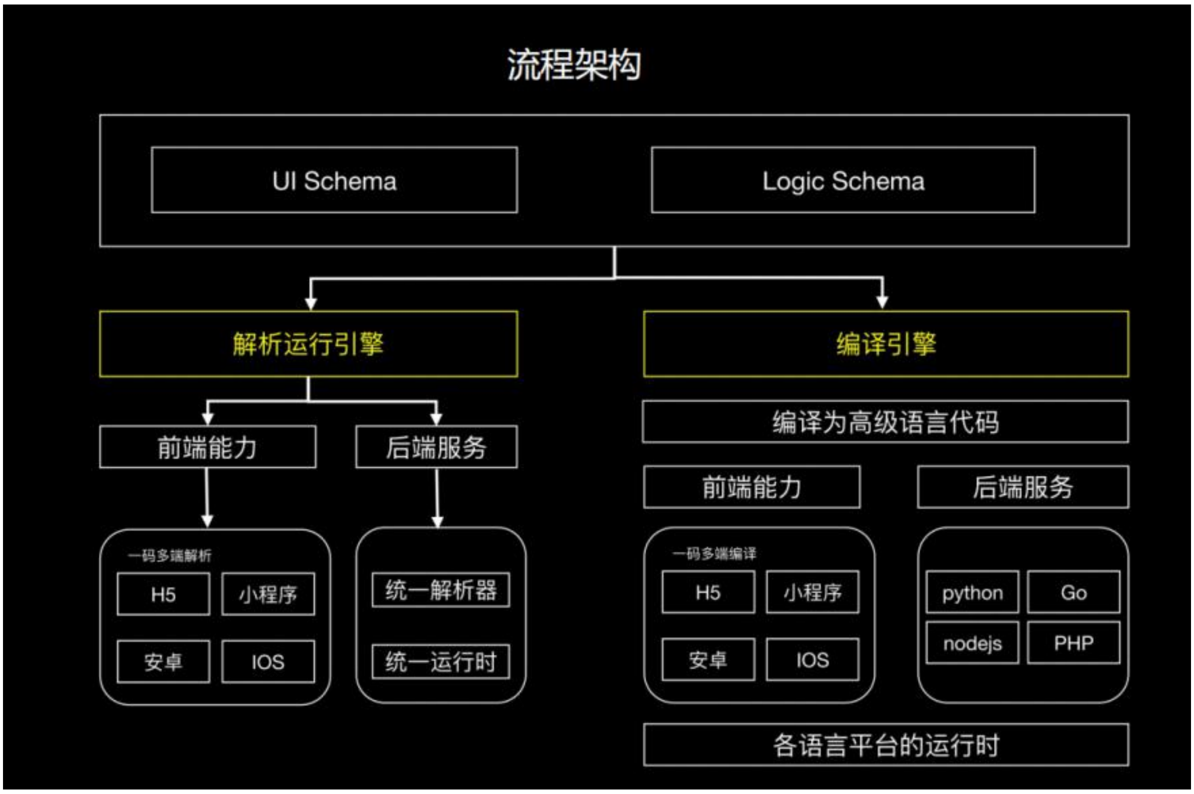
DSL Engine:

- Standard：文档定义（语法、语义）；
- Core：SDK 能力（To Schema、To AST、To DSL）；
- Render：插件能力（语法检查/高亮、语法提示）；

4、生产与运行



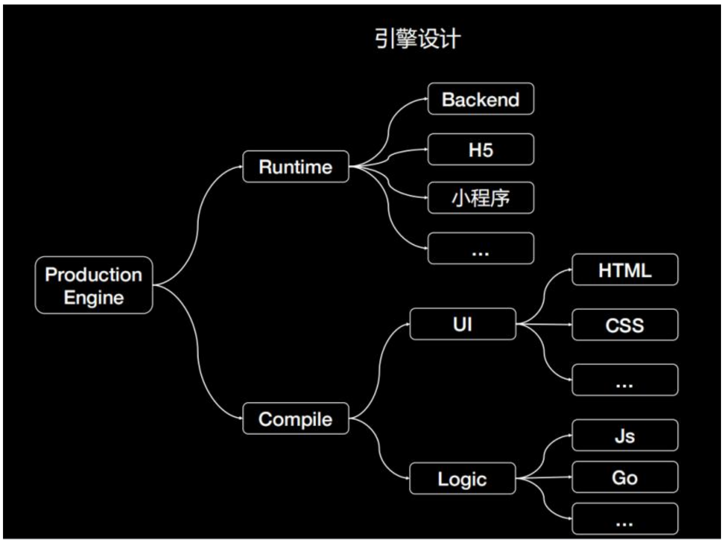
流程架构



UI Schema 和 Logic Schema 开发完之后，怎样把它变成实际产物？可以通过两套引擎：解析执行或编译执行。

- 选择解析运行引擎，前端服务和后端服务分开，前端能力可以进行解析，后端服务会统一一套解析器，会统一一套运行时；
- 选择编译引擎，通过编译引擎转换成高级语言代码，前端转化后在各自的语言平台上运行，后端转化后也是在对应语言的运行时去运行。

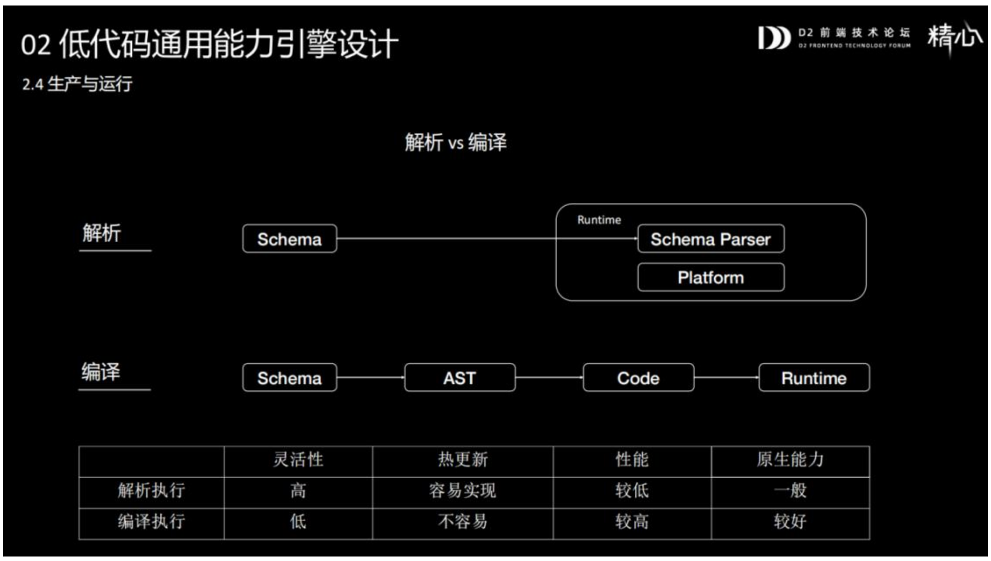
引擎设计



Production Engine:

- Runtime：Backend、H5、小程序；
- Compile:
  - UI：HTML、css；
  - Logic：Js、Go；

解析 VS 编译



- 解析的过程：从 Schema 直接到 Runtime(Schema Parser)
- 编译的过程：从 Schema 转化成对应语言的 AST,然后由对应语言的 AST 转化成对应的 Code 语言，然后在 Runtime 里执行

解析执行与编译执行对比：

	灵活性	热更新	性能	原生能力
解析执行	高	容易实现	较低	一般
编译执行	低	不容易	较高	较好

5、质量保证



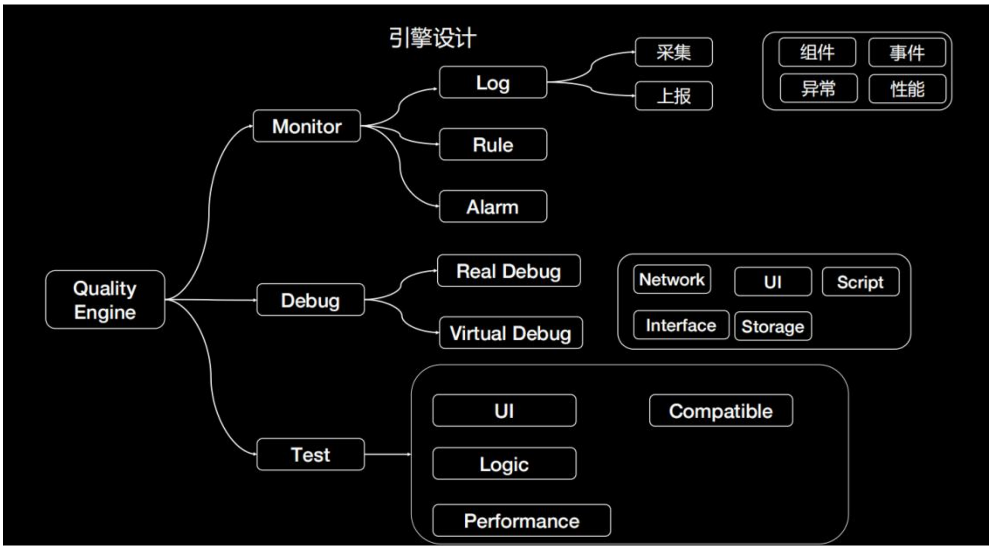
关键词

质量保证关键词：性能、调试、测试、异常上报、告警监控、健壮。

核心能力

	能力内容	特点	技术方案
低代码监控引擎	日志采集、日志上报、数据分析、告警与工单系统	纯配置化、多维度日志、按模块/组件配置、按事件配置	将日志采集配置低代码化 集成告警规则、告警引擎 整个 LCDP 基于 schema,更方便埋点和配置信息采集
低代码调试引擎	前端 UI 调试、后端接口调试	实时模拟器调试、真实环境调试、接口调试可视化	基于解析引擎，模拟器可以实现调试 基于 wesocket,编译成源代码在远端机上调试，实时返回调试界面与交互
低代码测试引擎	单组件测试、整体 UI 交互测试、前后端联合测试、压力测试、性能测试	测试用例可视化开发、灵活配置输入输出	设计一套测试逻辑解析引擎 各种外部因素实现可配置

引擎设计

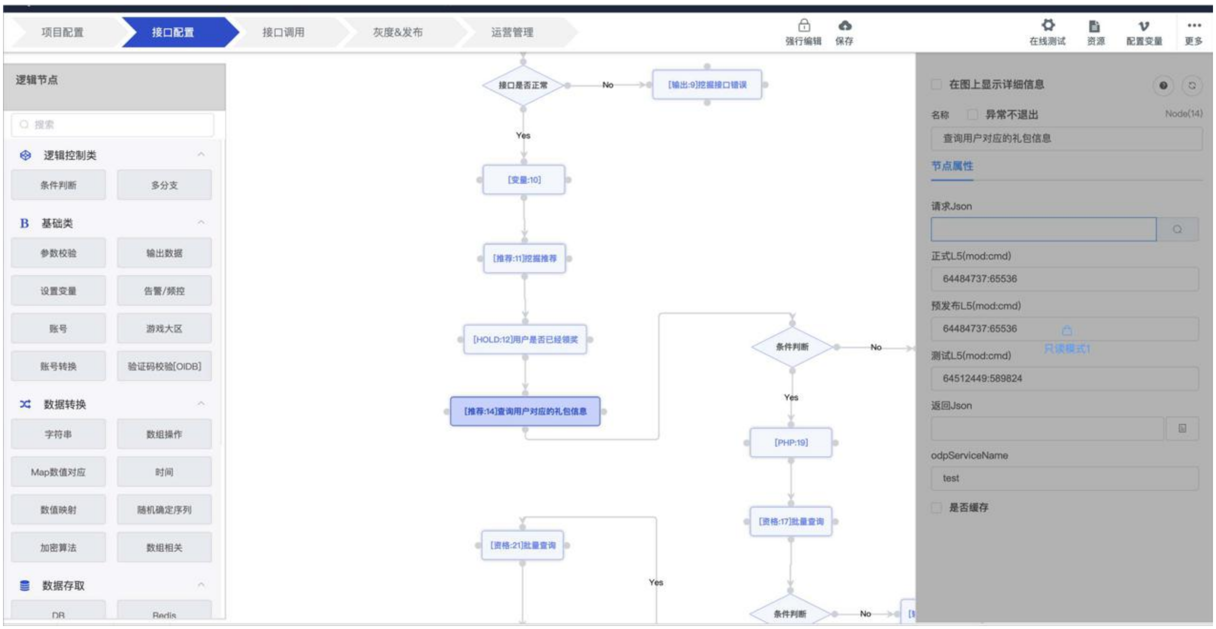


Quality Engine:

- Monitor：Log(采集、上报)、Rule、Alarm；
- Debug：Real Debug、Virtual Debug；
- Test：UI、Logic、Performance；

三、公司内低代码平台的实践

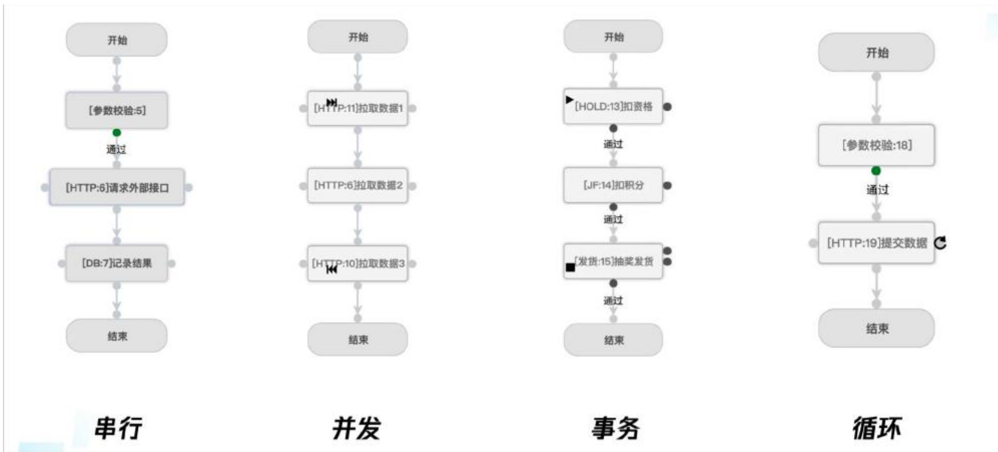
1、后端逻辑编排



上图是腾讯星图低代码平台，它的应用领域包括：互娱游戏营销活动后端逻辑开发、抽奖、活动签到、游戏预约、礼包领取等。

核心能力包括：代码级流程绘制；丰富的组件模块；可视化与代码的融合；便捷的接入调用；在线测试等。

代码级流程绘制：



丰富的组件模块：



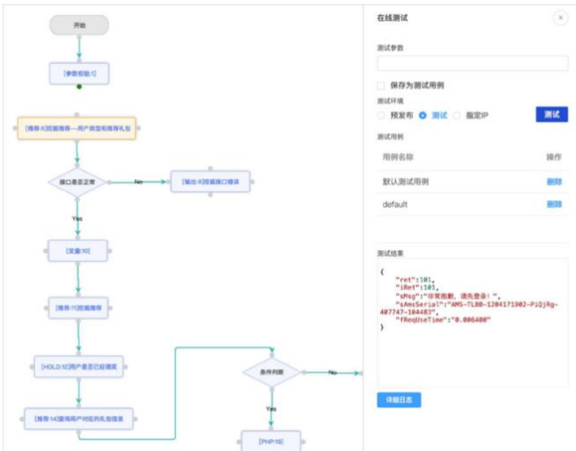
可视化与代码的融合：



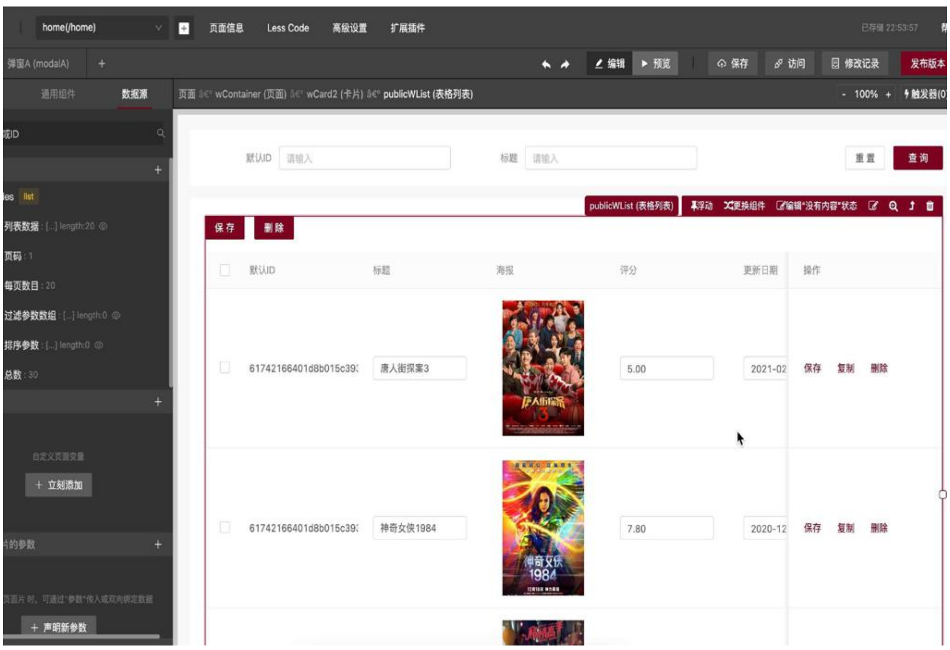
便捷的接入调用：



在线测试：



2、toB 中后台管理系统



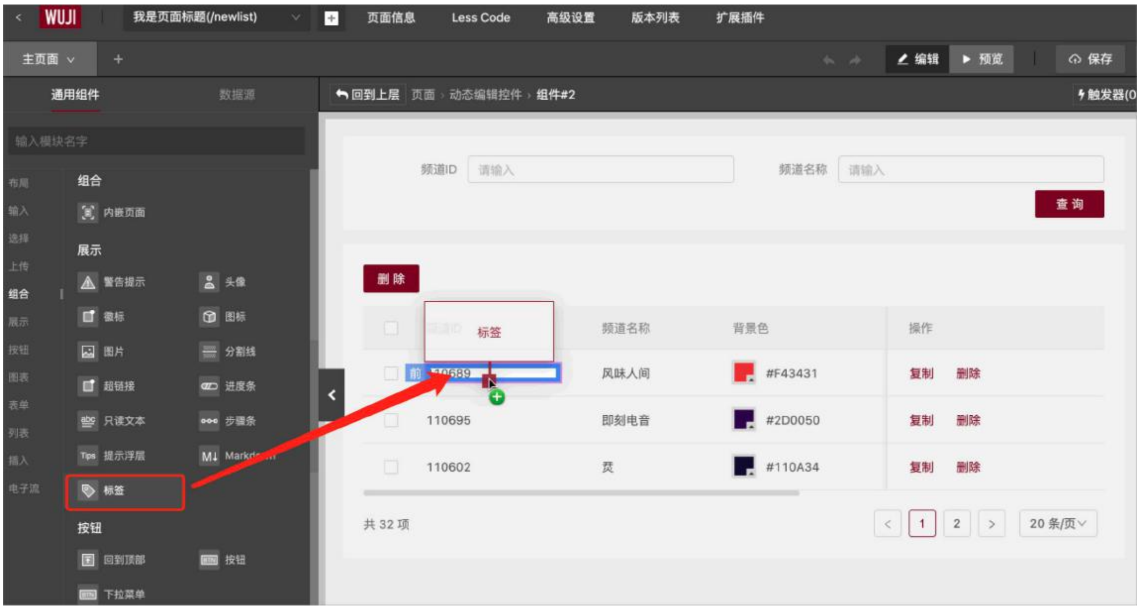
这是一张腾讯无极低代码平台的截图，它是公司内多款业务产品的管理系统的搭建。

核心能力包括：页面片的概念、灵活的画布、灵活丰富的数据源支持、本地 cli 模式。

页面片：



灵活的画布：



灵活丰富的数据源支持：



本地 cli 模式：



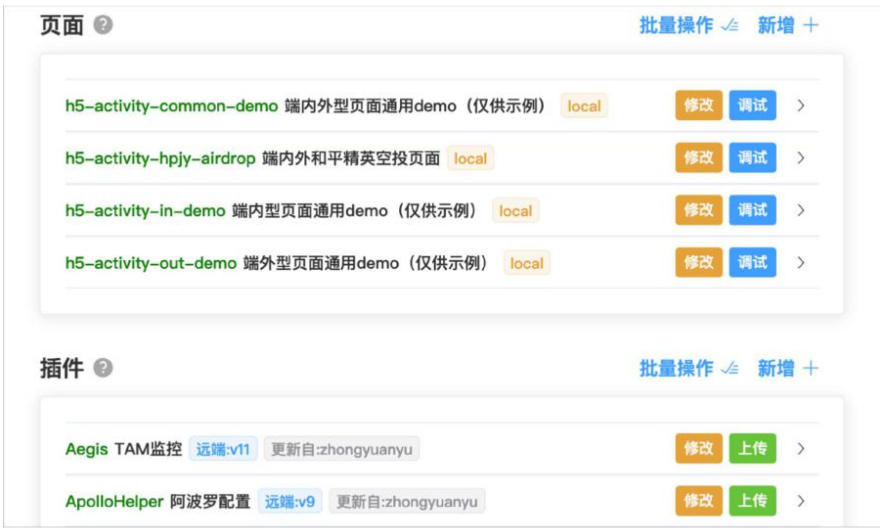
3、toC 的营销活动低代码



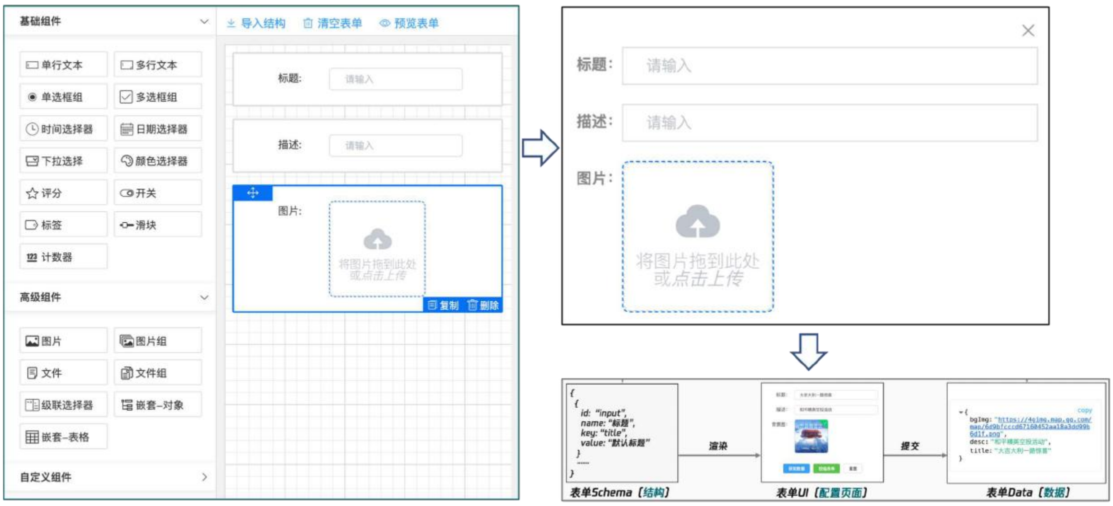
以腾讯地图雅典娜低代码平台举例，雅典娜是腾讯地图 toC 的营销运营活动的快速搭建平台，它主要是手机上的腾讯地图营销活动开发。

核心能力包括可视化物料生产管理、物料数据表单化定义、丰富的业务组件支持。

可视化物料生产管理：



物料数据表单化定义：



丰富的业务组件支持：



四、总结

1、低代码应该是一个生态系统；

2、LCDP 底层引擎的建设就像空气、水分、阳光；

3、最适合解决具体领域问题的 LCDP 就是最好的。

ESM Bundleless 在低代码场景的实践

摘要：近两年来，随着 Vite/Snowpack 等基于 ESM 的构建工具出现，给 Dev 研发带来了数十倍的提速。借助浏览器 ES Module import 的能力，特别是当 HTTP/2 支持请求与响应的多路复用的传输效率提升，我们可以不再需要等待一个完整 bundle 的构建。 蚂蚁集团的低代码搭建平台「云凤蝶」也很早在资产体系内建设了一套 Bundleless 加载方案，结合如今最火热的 ESM 方向和原有方案的痛点，我们重新设计了 Bundleless V2 方案，通过对依赖的分析及合并计算、二次构建的 Treeshaking 以及 import-map 将 ESM 真正部署到了生产环境，并在蚂蚁集团内部上百个中后台搭建场景中落地。

分享人：严铭(亦逊)，蚂蚁集团高级前端开发工程师，目前主要负责蚂蚁集团低代码搭建平台「云凤蝶」的资产 Bundleless 体系设计。

今天分享的主题包含三个关键词，ES Module，Bundle，低代码。或许你使用过或者听说过，甚至开发过一个低代码的平台。低代码的核心就是通过可视化或者配置的方式，极大的提升应用的研发效率，这也是我所在的团队“云凤蝶”这两年正在努力做的事情。

在蚂蚁集团，我们有超过 36%的中后台应用都使用云凤蝶搭建。在云凤蝶，我们提供了丰富的 UI 资产；自建可视化的画布，让用户写更少的代码来完成交互；对接集团的 API 元数据，一键生成表单和表格；甚至我们还提供了官方的存储和模型服务。

今天主要结合云凤蝶的实践经验，我们来聊一聊，对于一个企业级的中后台低代码平台，背后的资产加载体系是如何设计的？

一、背景

1、低代码组件

首先介绍一下云凤蝶开放的资产体系。那在设计一个搭建系统之前，首要的问题就是组建从哪里来？一套组件库的研发成本是很高的。以 Ant Design 为例，单个组件的研发加测试成本至少

在两周。

而在体验技术部，我们已经拥有了全球最好的一系列组件库，包括 Ant Design，Ant V，要是低代码平台能够直接拿过来用，就能够节省出非常多的时间和精力。

那么组件怎么拿过来用呢？常规有两种方案：

- 一种是平台集成这些组件库，直接管理组建的版本和打包构建；
- 另一种方案是通过特定的组件、协议和规范，能够实现组件库的动态加载；

那在实际的场景中，无论在标准的一个中后台应用，业务中往往都会有自己的组件沉淀和能力需求，这些都无法内置到平台。所以，低代码的资产体系必须要有开放的能力，将资产与平台解耦，然后再通过规范进行连接，这样才能打造出富有生命力的资产体系。

2、NPM 导入组件

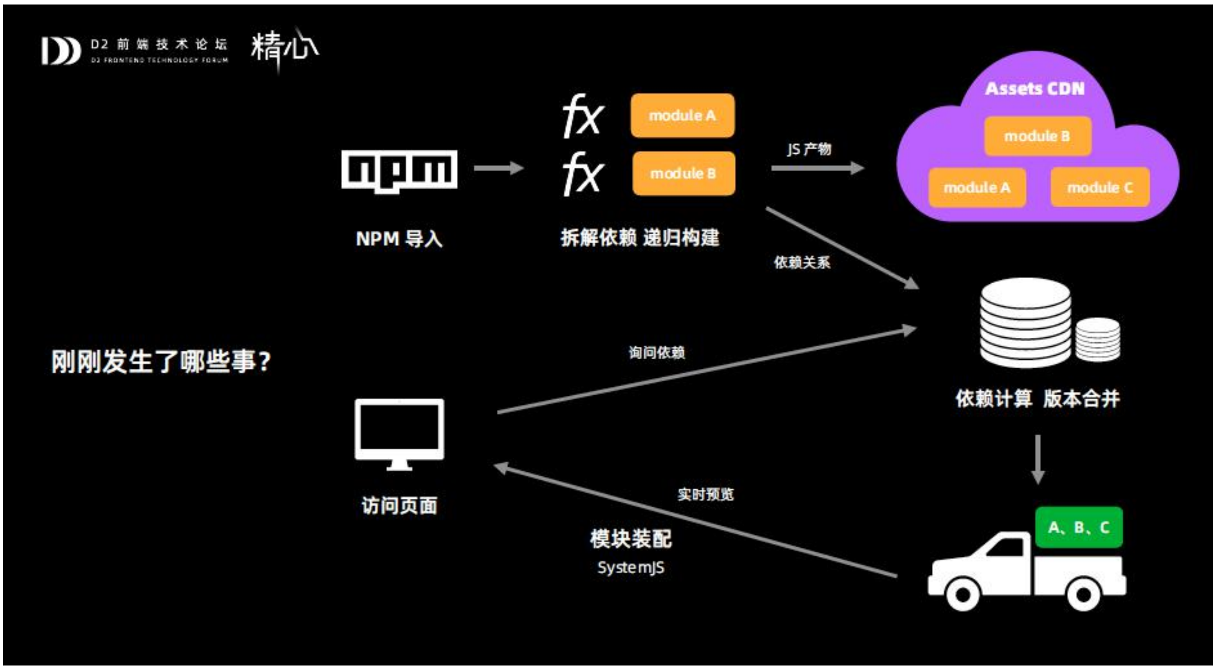
作为前端工程师我们知道，NPM 社区是一个非常有活力的社区，所以，我们确立了让组件进来的桥梁：NPM 导入。

NPM 导入低代码组件：

- 首先在云凤蝶 NPM 导入的面板中输入 NPM 包名和版本号后，就会自动解出 NPM 包导出的 react 组件并加载；
- 经过简单的配置，就可以在云凤蝶的可视化画布中添加这个组件，并且进行属性编辑了。

整个过程就会像 NPM install 一样丝滑。

导入过程

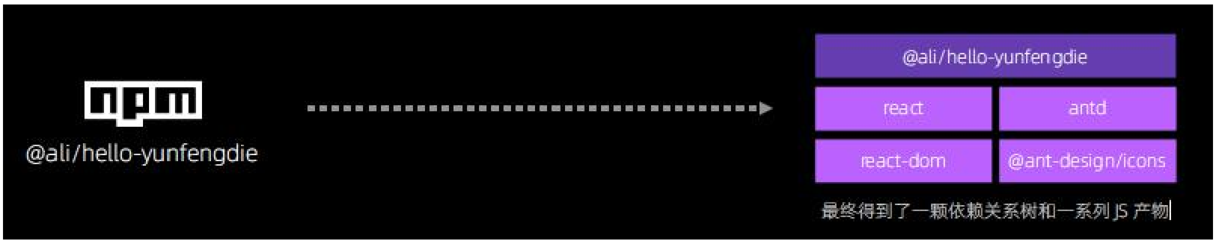


在 NPM 导入过程中，我们会对 NPM 包进行依赖拆解和二次构建，将依赖关系和构建好的 JS 产物上传到 Assets CDN 上，每次用户在云凤蝶编辑器中进行编辑时，我们都会处理最新的依赖关系，并根据版本进行依赖合并。

到了实际的预览阶段，将将合并好的依赖关系进行解析，然后将所有模块的 JS 产物一次性加载回来，交给 SystemJS 进行模块的实例化。

依赖分析和二次构建怎么做的?

- 从 npm registry 下载 @ali/hello-yunfengdie;
- 依据 package.json 浅层分析依赖;
- 下载依赖，递归执行 1;
- Webpack 构建，external 子依赖;



最终一个 NPM 包得到了一颗依赖关系树和一系列 JS 产物，基于此，就完成了 Bundleless 资产体系 1.0 的建设。

3、Bundleless 资产体系 1.0

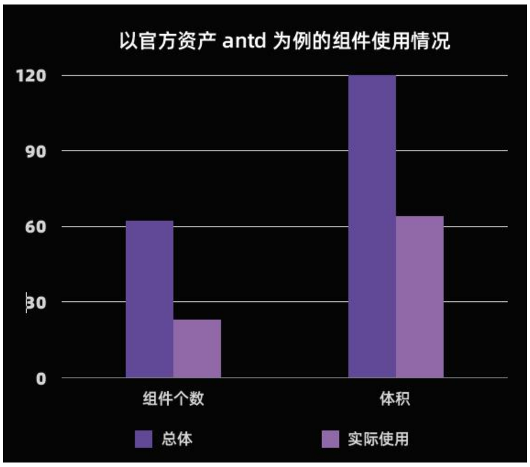
- 完全对接 NPM 生态：丰富的资产体系，拒绝重复建设；
- 依赖复用：遵循 Semver 规则的依赖合并；
- 按需加载：包级别的按需加载；
- 实时预览 & 调试：秒级部署上线支持 debug 调试；

4、存在的问题

这套方案支撑了云凤蝶上百个中后台的应用，但在实际的发展过程中，我们也慢慢的发现了一些问题。

业务上的诉求：不够快，资产加载成为性能瓶颈

以官方资产 Ant Design 为例，官方一共提供了 62 个默认好用的组件，但在实际中后台搭建场景中，大部分的站点使用个数只是 25 个，也就是在整体数量的 40%，而我们选取的使用频率最高的 25 个组件进行的 Boundle 测试，大约是 640KB，而 Ant Design 整体体积是 1.2M，也就是说，差不多有一半的冗余体积。

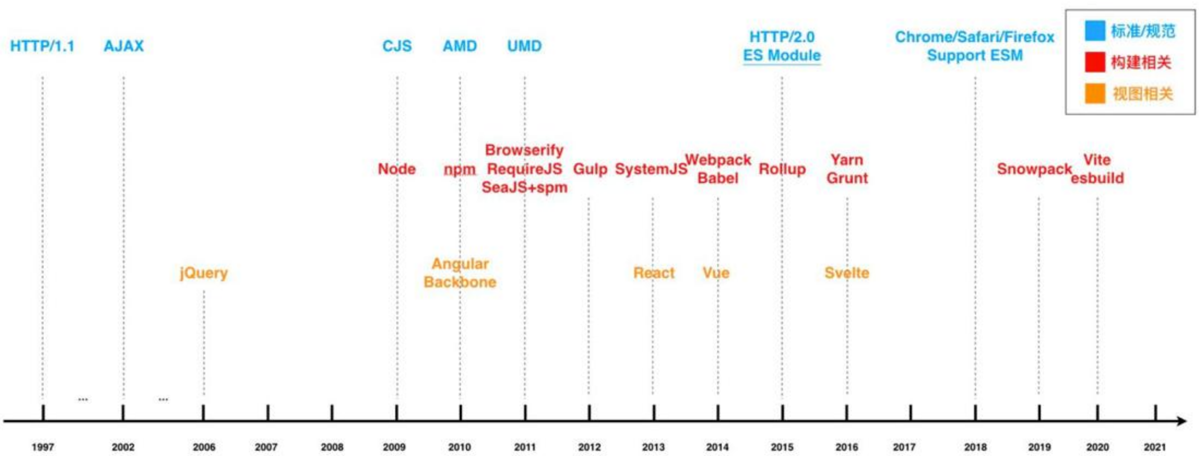


在搭建场景中，包级别的按需加载还远远不够，细粒度到组件级别的按需加载才是更适合搭建的场景。

此外，受限于方案中的 SystemJS 的性能，会影响模块实例化的耗时，我们也在去寻求是不是会有更快的模块的加载方案。

技术上的追求：不够 modern，无论是模块规范还是加载方案

我们回头去看过去 20 年前端发展的路线会发现，视图相关从 jQuery 一直发展出了 react、Vue 等一系列优秀的框架；构建工具也从 Webpack 到近两年大家所熟知熟知的 Vite、esbuild，这一切的背后，还伴随着网络能力、浏览器标准和规范的升级，从 HTTP1.1 到 HTTP2.0，从 2G、3G，到如今的 5G，从 CommonJS 到如今标准的 ES Module，如果我们把刚刚介绍的云凤蝶的资产加载体系放在这条实践线上，我们会发现，差不多停留在五六年前，显然还不够 modern。



对下一代资产加载方案的憧憬

体积上的下降和性能上的提升，核心是设计一套组件级别按需加载、兼顾生产同时面向未来的高性能模块加载方案，于是，我们将目光看上了 ES Module。

二、ESM 的机会和挑战

1、应用现状

ESM 能够上生产了吗？它的浏览器兼容性怎么样？性能如何？能不能解决当前云凤蝶遇到的问题？

除了 IE11 之外，目前 Chrome、firefox、Safari 的支持度都比较友好，可以完美的支持中后台业务场景下的浏览器兼容性要求。

同时，在单模块的加载性能上，对比我们之前的 SystemJS，有 1.3 倍到 1.5 倍的提升。

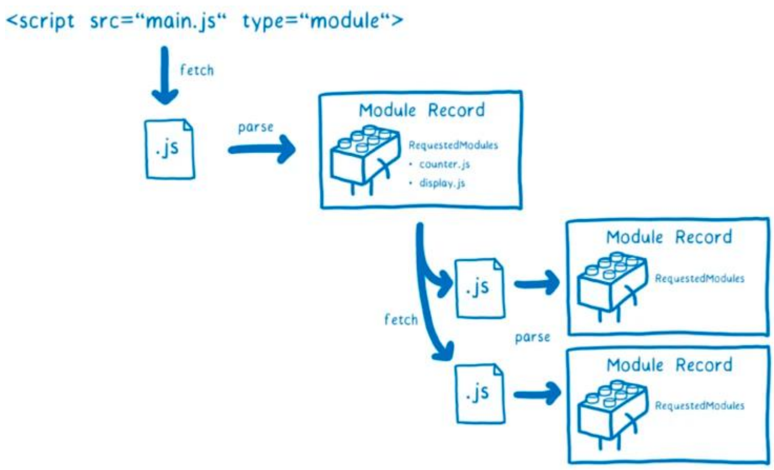
2、工作原理

有了兼容性和性能上的保障之后。我们再去看看 ESM 究竟是如何工作的。

浏览器从 ESM 入口文件开始解析，找到每一个导入语句的模块说明符，下载这个文件，继续解析，直到所有的文件解析完成之后，通过深度优先的后续便利，去进行模块的实例化。

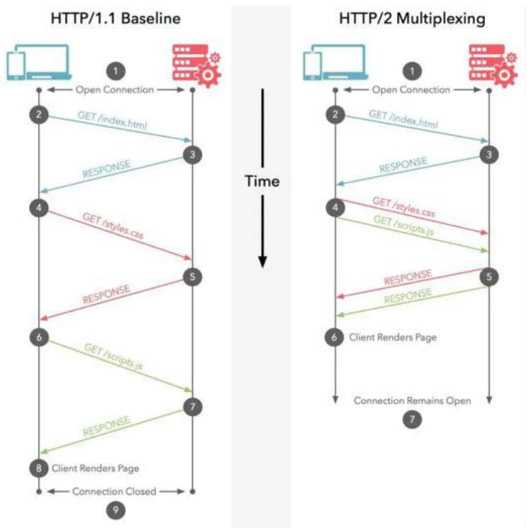
请求瀑布问题

在这个过程中，递归的模块解析和下载会带来请求瀑布的问题。并且。由于 ES module import 和 export 的顺序要求，我们没有办法像 UMD 的模方案一样去做 CDN 的 combo。而如果依赖关系一旦复杂，那么不管单模块下载有多块，整体的体验和速度都不够理想。

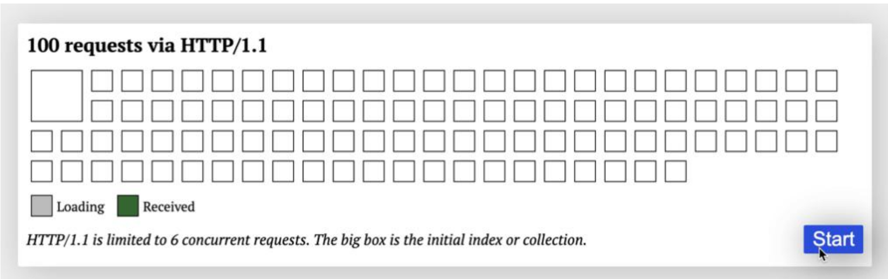


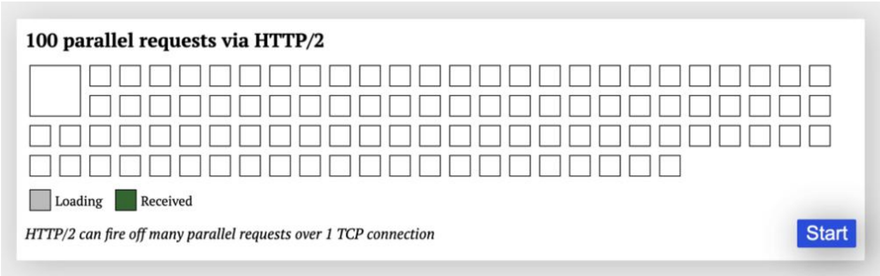
HTTP2.0

那在 HTTP1.1 中，浏览器在同一时间内针对同一请求，是有一定限制的。以 Chrome 为例，最大的并发限制是 6 个，而 HTTP2.0 多路复用则允许通过单一的连接进行双向的交换。



HTTP2.0 把通信的基本单位缩小为一个个的祯，并行的在同一个 TCP 连接中进行双向交换。





HTTP2.0 对比 HTTP1.1，在性能上，尤其是高并发的场景的提升是非常非常明显的。但是我们都知道 JS 的模块的依赖复杂度是很高的，大家可能都听说过的一些 Node modules 的黑洞。所以我们认为。HTTP2.0 或许只能说是 ESM 上生产的必要条件，但并不是标准答案。HTTP2.0 帮助我们去解决的并发的问题，也就是依赖树广度的问题，但是依赖树的深度还是没有解决。

V8 的建议

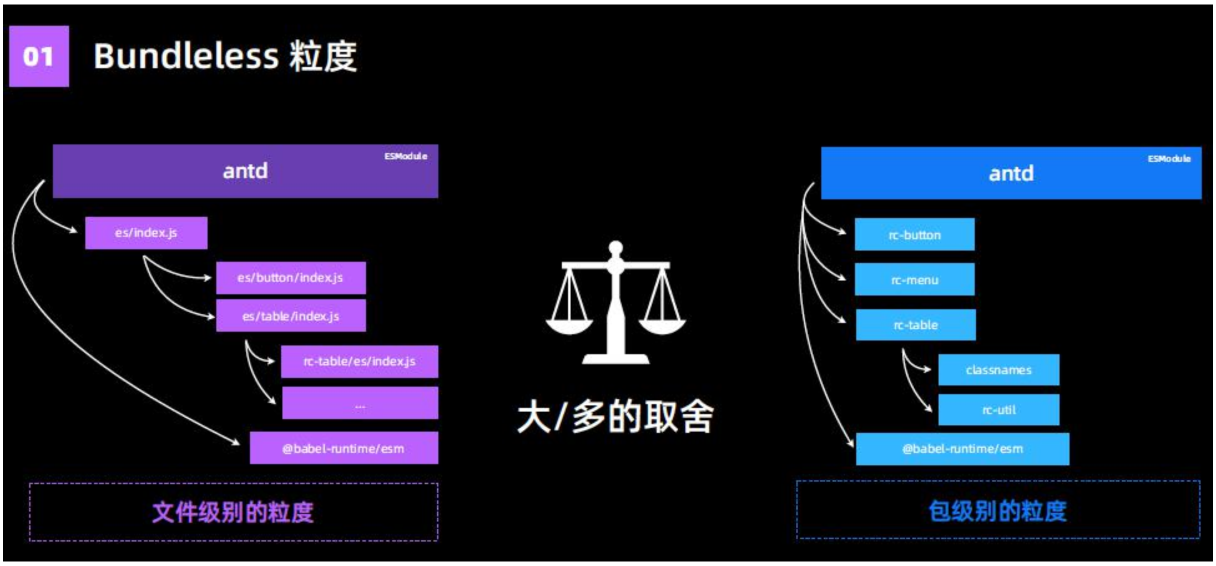
V8 在 2018 年对使用 ES Module 的性能建议，可以在以下两个场景去使用 ES Module，而无需借助 Webpack 等打包工具进行 Bundle：

- 第一个是本地研发。像近两年火热的 Vite、Snowpack，通过本地的对依赖的预编译 ESM，和本地文件的单文件 ESM，极大地提升了本地研发的效率；
- 第二个是生产环境，推荐小型 Web 应用。V8 在这里也对小型 Web 应用作了解释：模块的整体数量小于 100，依赖层级小于 5。

那么距离 2018 年过去已经三年了，低代码搭建可以吗？严格意义上说，任何的中后台应用，它都算不上一个小型外部应用，一个中后台应用有二、三十个页面，四、五十个依赖，我怎么可能算一个小型的 Web 应用呢？但是我们单拎资产加载，或者说云凤蝶的资产加载来看，通过对云凤蝶资产的历史数据分析，我们认为，通过特定的一些优化策略去收敛这些复杂度，达到 V8 的要求是具备可行性的。

三、ESM Bundleless 设计思路

1、Bundleless 粒度



Bundleless 粒度决定了模块的数量级，依赖树的复杂度，以及依赖复用程度，拆分的粒度越大，模块数量越多，整体的复用程度越高，但体积会越小；拆分的粒度，越粗，模块的数量会越小，复用程度会越低，但整体的体积会越大。

所以，拆分的粒度是体积和数量上的权衡。经过我们的测试，相较于文件级别的海量请求，包级别的粒度会更适合生产环境。

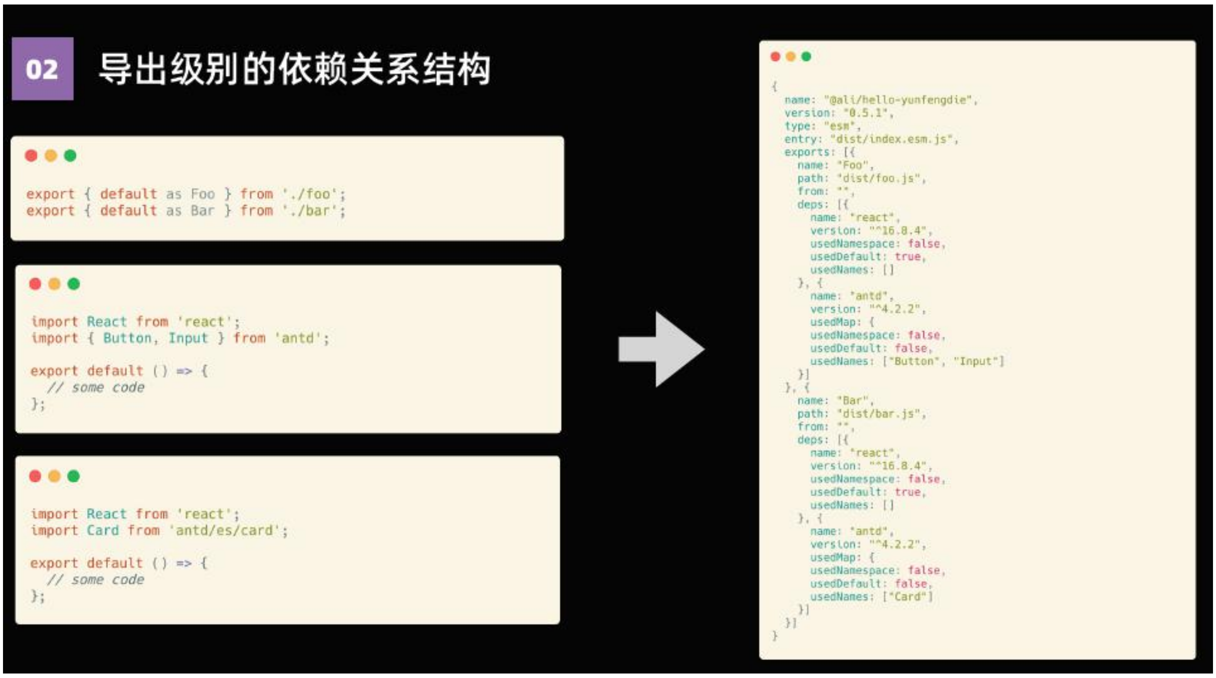
2、导出级别的依赖分析



有了包维度拆分依据之后，可以分析出每个 NPM 包的依赖关系。之前是按照包维度做依赖分析，但在这一次的方案中进一步做了导出级别，因为在搭建系统中，组件级别按需加载才更适合。而且组件级别的，TreeShaking 在 Bundleless 方案中尤为重要。它不仅降低了整体 Bundle 的体积，还因为摇掉了一些不必要的依赖，从而显著的降低依赖树的复杂度。

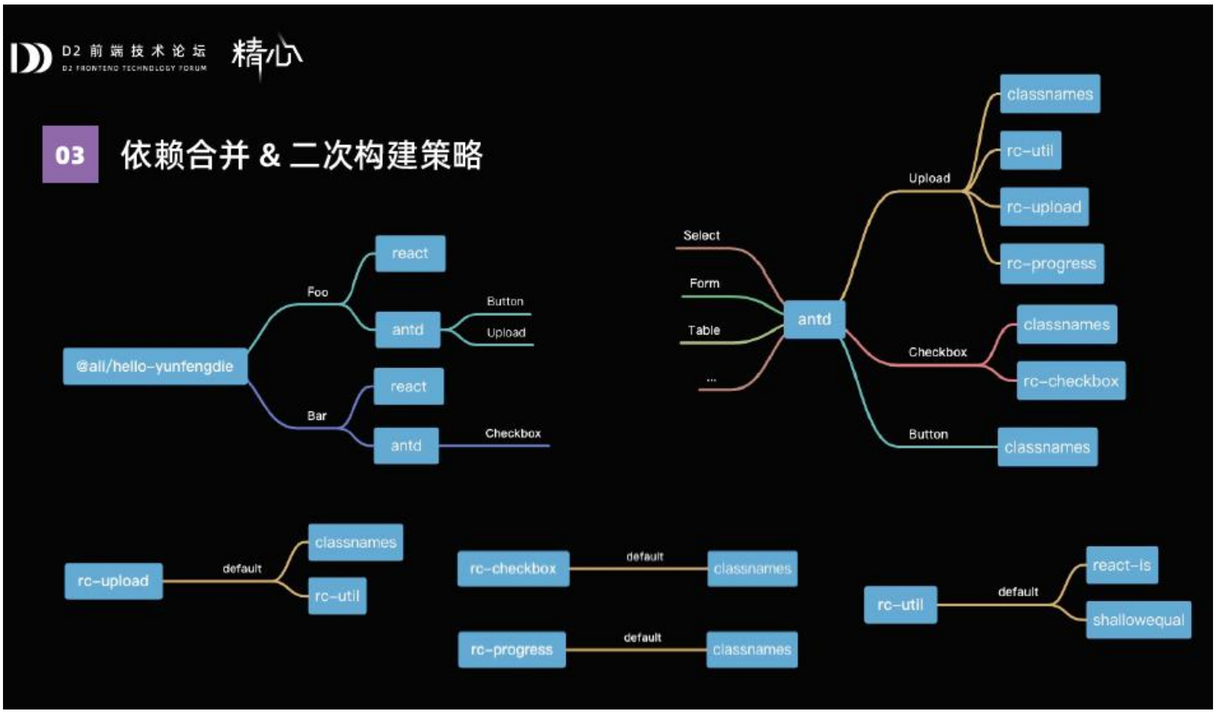
我们通过 AST 去分析每个 NPM 包的导出，然后对每个导出去做了独立的依赖分析。，MPM 包所有导出的依赖关系。

以下图左侧的代码为例，就可以得到右侧的结构化的导出级别的依赖关系结构，

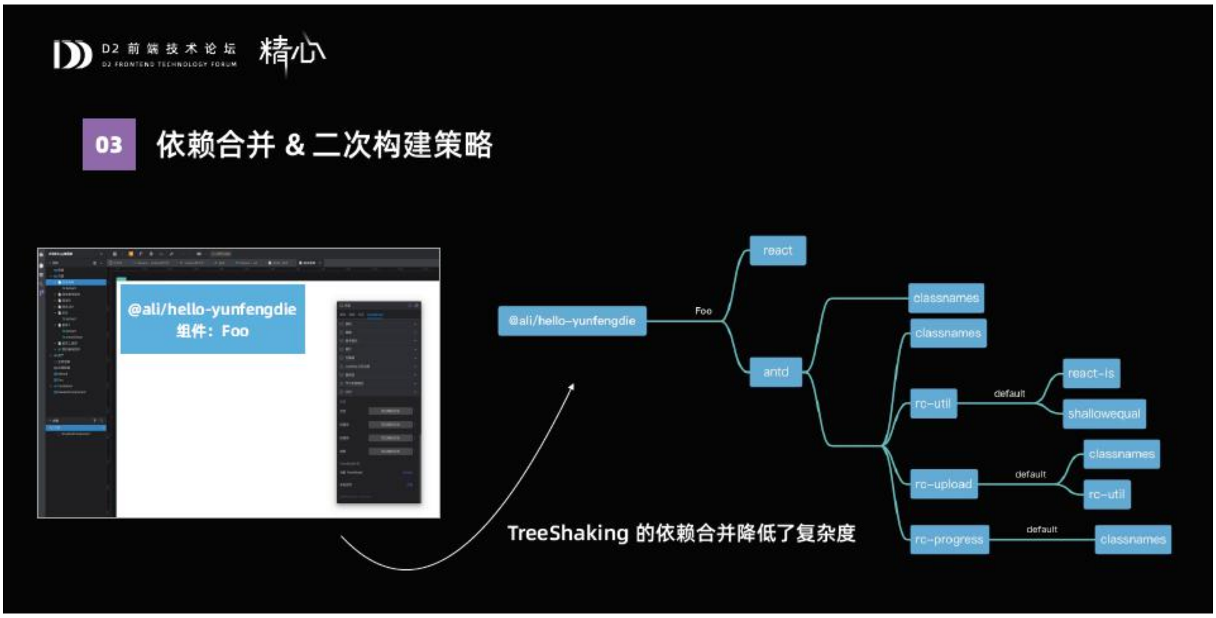


3、依赖合并 & 二次构建策略

我们已经得到了所有 NPM 包导出级别的依赖关系：

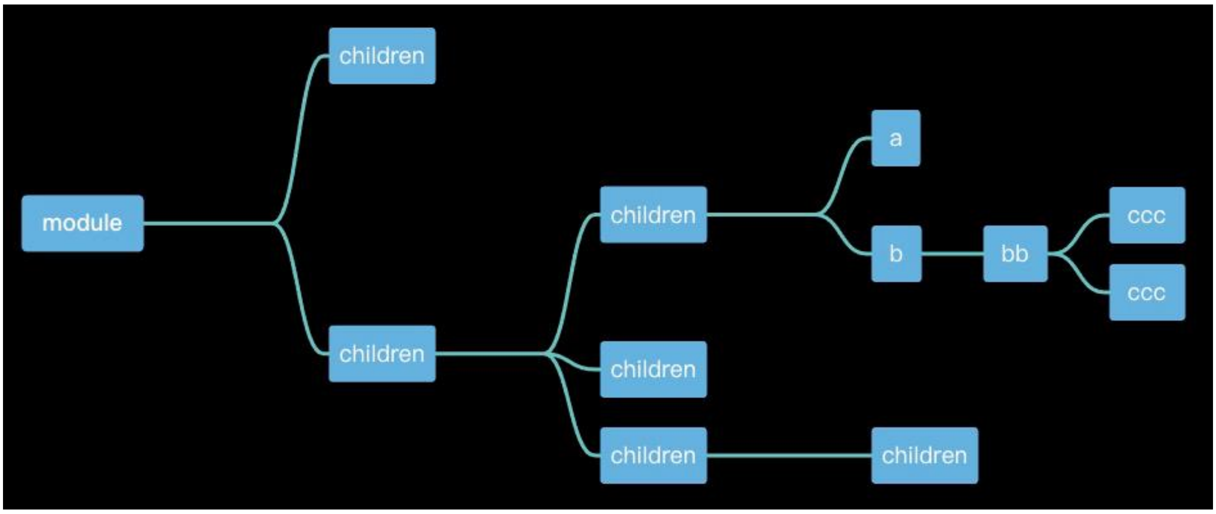


紧接着在云凤蝶，我们就可以根据实际的组件消费情况，进行导出级别的依赖关系合并，得到一颗 TreeShaking 后的依赖关系树，依赖关系结构的复杂度已经显著的降低。



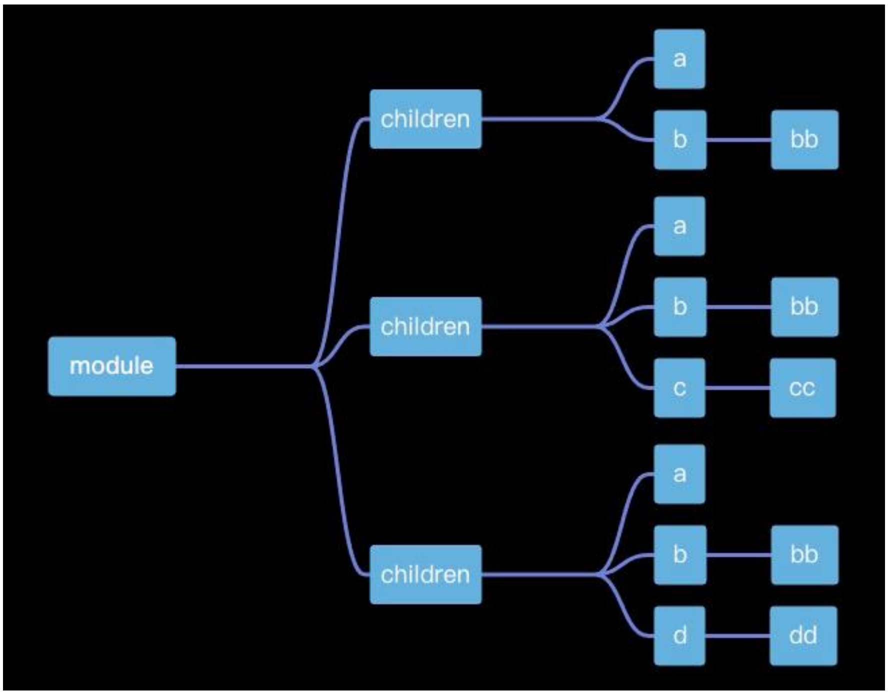
但是在实际情况中，我们还会遇到更复杂的一些依赖结构。这里我举两个例子。

首先是一个孤立节点的问题，如下：



e.g.「孤立节点」可以直接 Bundle

当一棵树中的某一个节点，它的子依赖嵌套非常复杂，但是它的所有的子孙依赖都没有参与过整棵树的依赖合并时，这个节点就可以被立为孤立节点，可以直接 Bundle 在一起，不拆分依赖。



e.g. 所有导出的公共依赖可以 Bundle

那同样的，对于一个 NPM 包所有的导出，如果都依赖了两个以上的公共依赖，并且这两个公共依赖在整个依赖树的合并策略中，也没有进行过依赖合并，那这个公共依赖也可以适当的 Bundle 在一起。

4、import map 最后一公里

经过了复杂的依赖合并和适当策略的 Bundle，我们成功地将依赖关系的复杂度收敛到了数量级 100、层级 5 的要求，最后就是模块引入的路径问题。

ESM 在识别、引入文件时是只支持相对路径，也就是我们平常写的 import react from react，有两种方案可以解决这种问题：

一个就是显式的文件引用。我们将带有构建哈希的文件路径，直接写入到引用文件的 ESM 文件中，但是这样的问题就是。我 AntD 更新了，我所有引用 AntD 的 ESM 文件都需要更新，这样的缓存策略在上生产的时候是有很大的问题的。

```
<script type="module">
  import React from './v/react@16.14.0/index.mjs';
  import { Button } from './v/antd@4.16.13/index.mjs';
</script>
```

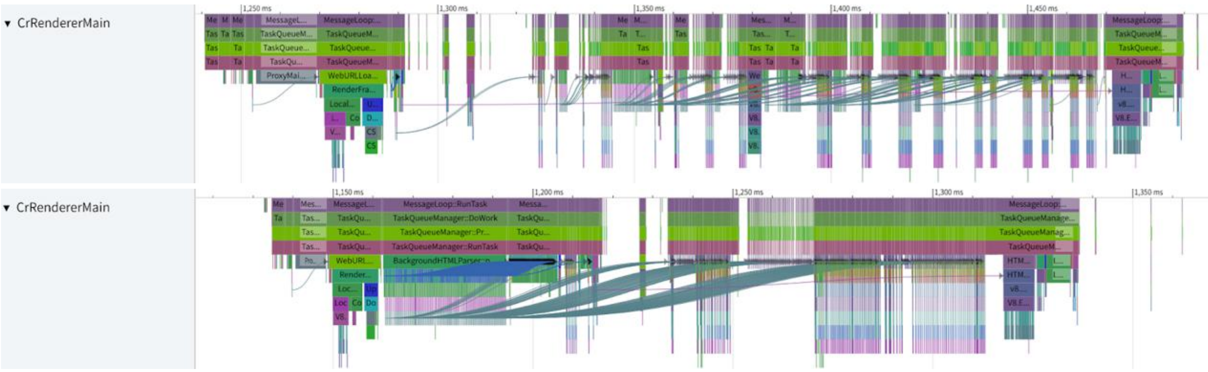
第二种方案是 import map，通过提前预声明依赖关系对应的地址和结构，浏览器就可以正确的识别出 react 和 AntD 具体的 ESM 文件地址，然后就可以正确的去下载。每当有依赖发生变化的时候，我们不需要去重新构建整颗依赖树，只需要去替换 import map，构建单独的那一个依赖就可以了。

```
<script type="importmap">
  {
    "imports": {
      "react": "./v/react@16.14.0/index.mjs",
      "antd": "./v/antd@4.16.4/index.mjs"
    }
  }
</script>

<script type="module">
  import React from 'react';
  import { Button } from 'antd';
</script>
```

5、预加载 modulepreload（额外 Buff）

我们还借助了 modulepreload 去提前预加载，深层次的子依赖，从而去大幅的改善我们整体的加载时间。这个经过我们的测试，大约是 1.3 倍到 1.5 倍的提升。



modulepreload 大幅度地改善加载时间（1.3x ～ 1.5x）

6、总结

我们通过 NPM 导出级别的依赖关系合并，以及 ES Module 自身的 TreeShaking，还有特定的适当策略的适当 Bundle，以及运行时模块预加载和 import map，成功的将云凤蝶资产加载体系搬到了 ES Module 的生产环境上。

四、未来展望

最后我们再去展望一下未来的模块开发方案和前端研发方式。

1、首先像刚刚一直在介绍的 ESM 的一些优势，它是社区标准，能够去更好的去做静态分析，以及 TreeShaking 的优势，未来社区上的 NPM 包都会优先去采用 ESM 的格式。

2、同样的，未来也会有更多 Bundleless，甚至 NoBundle 的构建方式的产生，我们可以更好的去享受浏览器标准升级、以及网络升级带来的一些优势。

3、此外，在目前 LowCode 的和 ProCode 的研发场景中，由于原先天然的两个平台的割裂，我们没有办法去做混合研发，只能借助 DSL 互转和微前端，但这两种方案都没有办法彻底解决两者依赖的复用问题。之后借助 ESM 统一的构建方式，我们可以天然的去解决依赖复用的问题。

感谢大家的聆听，下面是我的个人微信二维码，大家可以私下加我进行技术上的一些交流。



面向中后台复杂场景的低代码实践思路

摘要：中后台复杂的根源来自于逻辑成长带来的多状态黑盒，以及提升体验带来的高昂开发成本，本话题将讲述如何从偏重逻辑和偏重交互的两种不同类型的低代码方案去击破难点。

分享人：左俊(偏左)，阿里云产品商业化系统前端负责人，专注复杂中后台领域，致力于低无代码及不同领域的前沿技术提升复杂系统易用性解决方案。

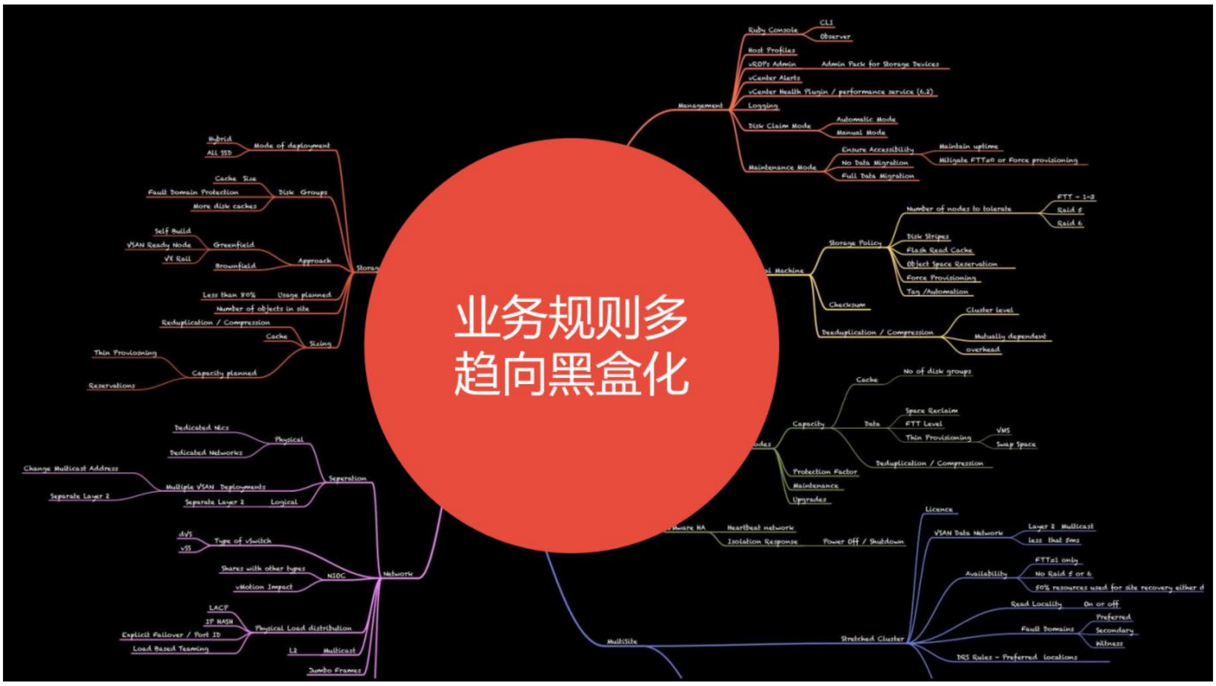
目录：

一、方案背景
二、动态标注生成交互界面
三、策略编排生成逻辑代码
四、Next

本篇内容将分享阿里云产品商业化系统做开发时的实践经验。此系统支持整个商业化售卖的完整流程。系统用到了我们所熟知的 ECS、RDS、OSS 等等。涵盖了很多重要的商业模式，需要很多配置环节。就复杂度而言，就超过了很多中后台系统。涉及到的问题同大部分中后台系统涉及问题基本一致。如何界定问题，分类问题，解决问题？

一、方案背景

1、理不清的业务规则，逻辑逐渐杂乱



做中后台开发经常会遇到复杂业务逻辑的问题，业务规则根据多人业务迭代逐渐变乱。现实中业务迭代频繁，场景多，速度快，某些业务存在严重的历史包袱。由此而来，作为前端开发，十分痛苦。

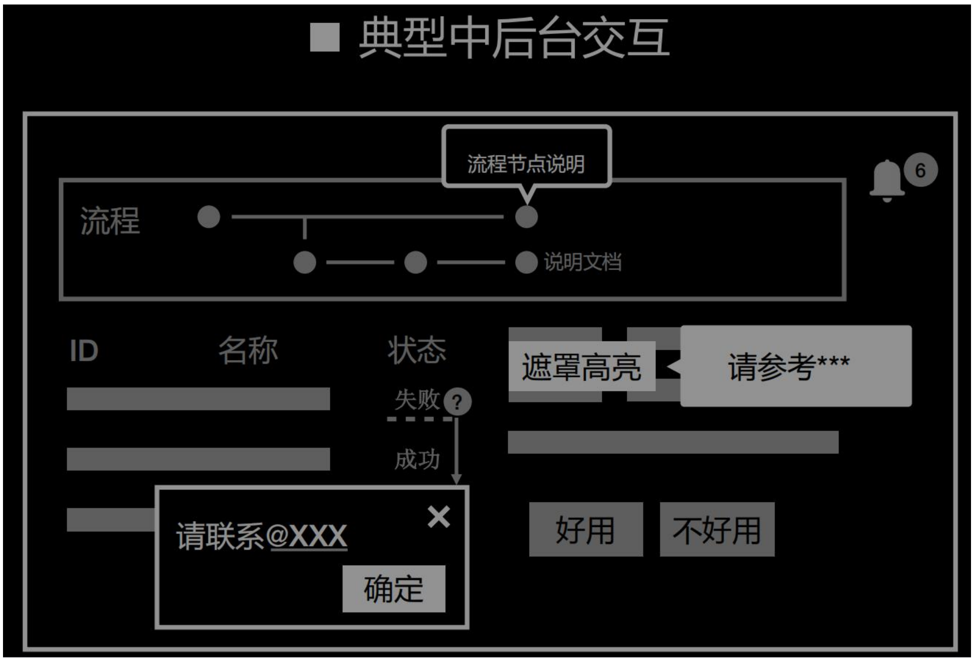
2、提示指引等易用性交互将开发复杂化

复制/粘贴加速腐化，容易干扰业务功能：

- 内容运营需要解释的内容多；
- 易用性交互形式多，并不限于图文；
- 易用性需求排期低于功能需求，平台越来越难用；

二、动态标注生成交互界面

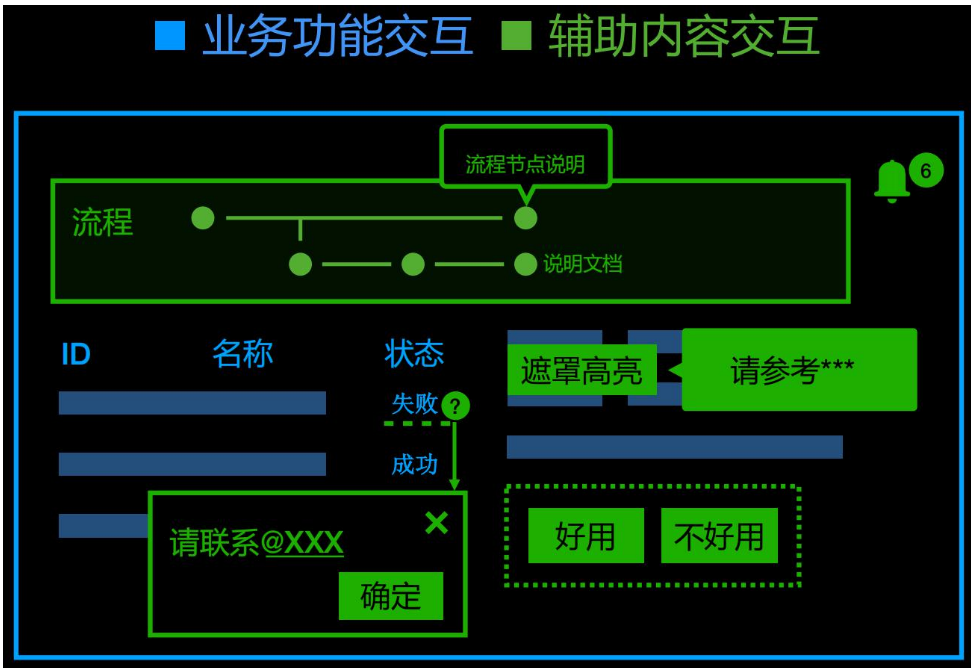
1、中后台页面示例



上图是一个典型的中后台交互页面，左边是列表，右边是明细，除此之外有很多指引提示性的内容，如流程图、下划线、气泡按钮等，而这部分提示性内容是可以通过低代码配置来完成的。

2、页面按功能划分

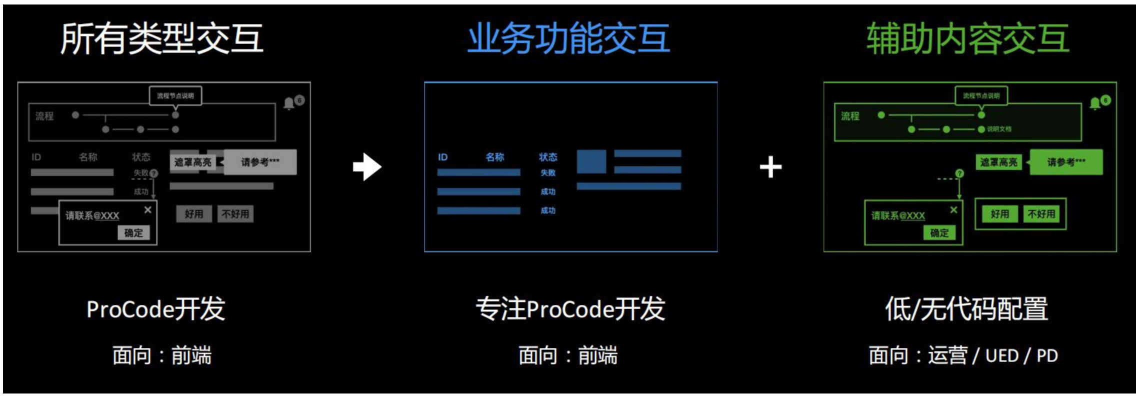
我们把上面页面按照功能划分，就分为了业务功能的交互和辅助内容的交互。



业务功能交互是指：业务如果脱离这部分交互，就是不完整的系统。

辅助内容交互是指：如果缺失了这部分交互，系统受太大影响，可以运行，就是用户体验度欠佳，增加了操作使用繁琐度。

3、方案核心目标



所以，我们方案的核心目标，就是将所有类型交互里面的业务功能交互专注 ProCode 开发；将辅助内容交互全部交给低代码配置去完成。

4、方案实施效果



演示系统中业务具体的实现效果如上图，涵盖了业务的功能以及其它的内容，都是辅助性的。如果将这近 400 个辅助交互都用 ProCode 开发实现，试想下开发和今后运营的工作量有多繁而杂？因此，我们选择用低代码配置来实现。

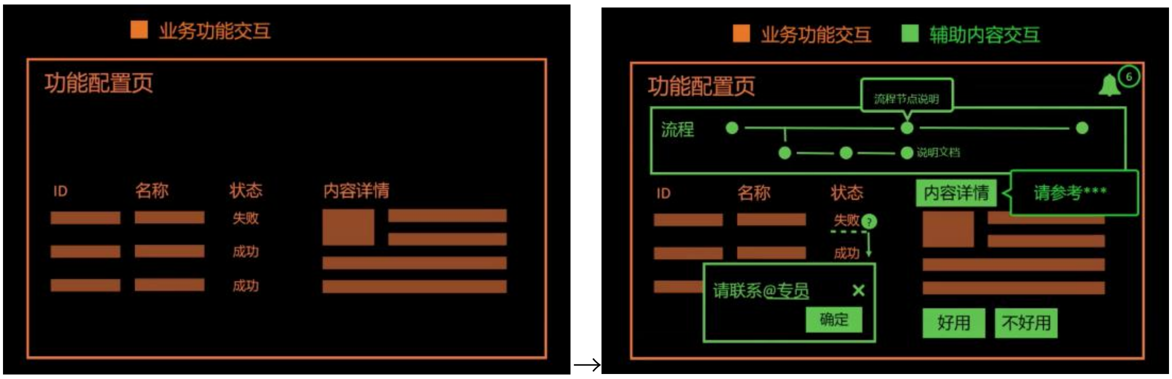
5、动态标注方案实践

步骤 1：分解交互界面，找到辅助类交互

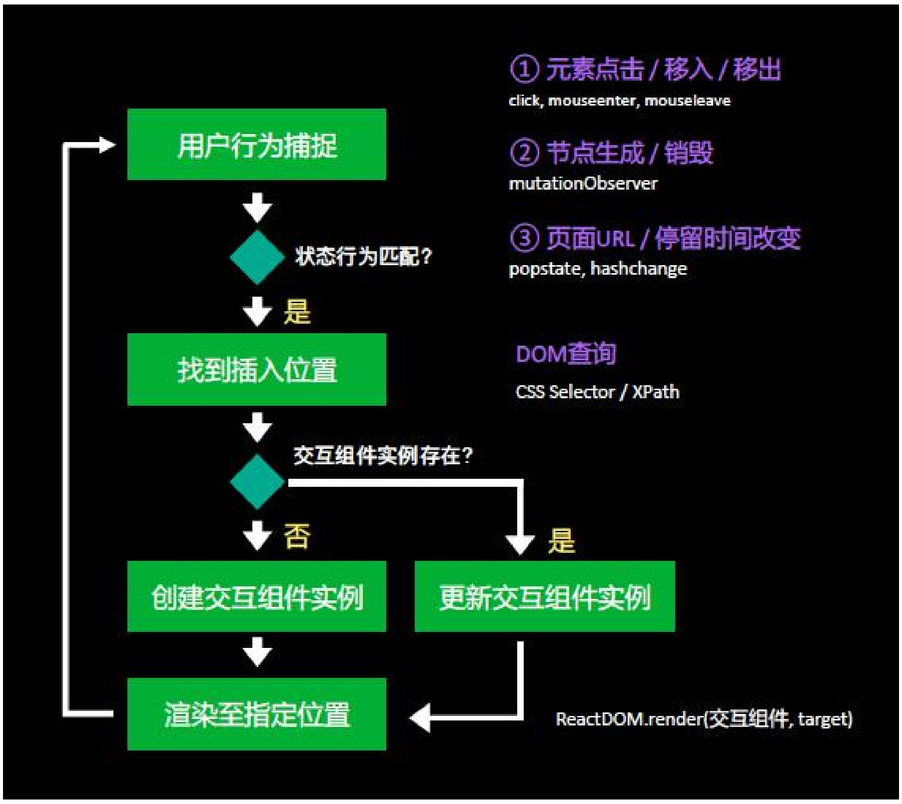


将经典的中后台交互拆分为业务核心功能交互和辅助内容交互，重点在于编排辅助类的交互，可以将此交互抽象为一个个可复用的组件。组件包含容器类、内容类、场景类、外部能力类等，其抽象过程是根据业务迭代逐渐沉淀成长起来，上图中右下角展示的都是实际业务中抽象出来的组件。

步骤 2：将交互组件按规则动态渲染到界面



上图左边是干净简洁整齐的页面，如何把刚刚抽象出来的可以复用的组件标注上去呢？

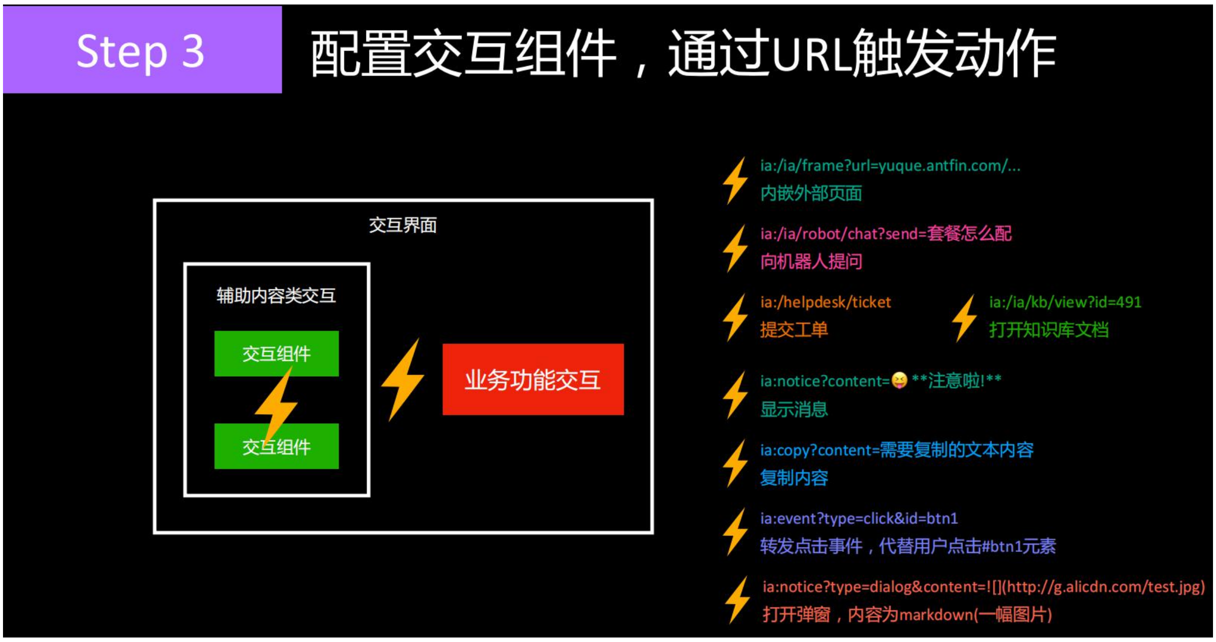


动态标注流程：

- 按照捕捉行为类型去捕捉用户行为；
- 捕捉行为类型包括：
  - 元素点击/移入/移出（click/mouseenter/ mouseleave）
  - 节点生成/销毁（mutation/Observer）
  - 页面 URL /停留时间改变（popstate/ hashchange）

- 找到确切的插入点，将组件插入到对应的位置；
    - DOM 查询，CSS Selecton / XPath；
- 如果该位置存在组件就进行更新，否则就创建组件；
- 渲染至指定位置。
- 标注流程并非一次性解决，可能页面上有很多的组件都是需要标准的，要设置很多规则，都需要按照动态标注流程标注上去。

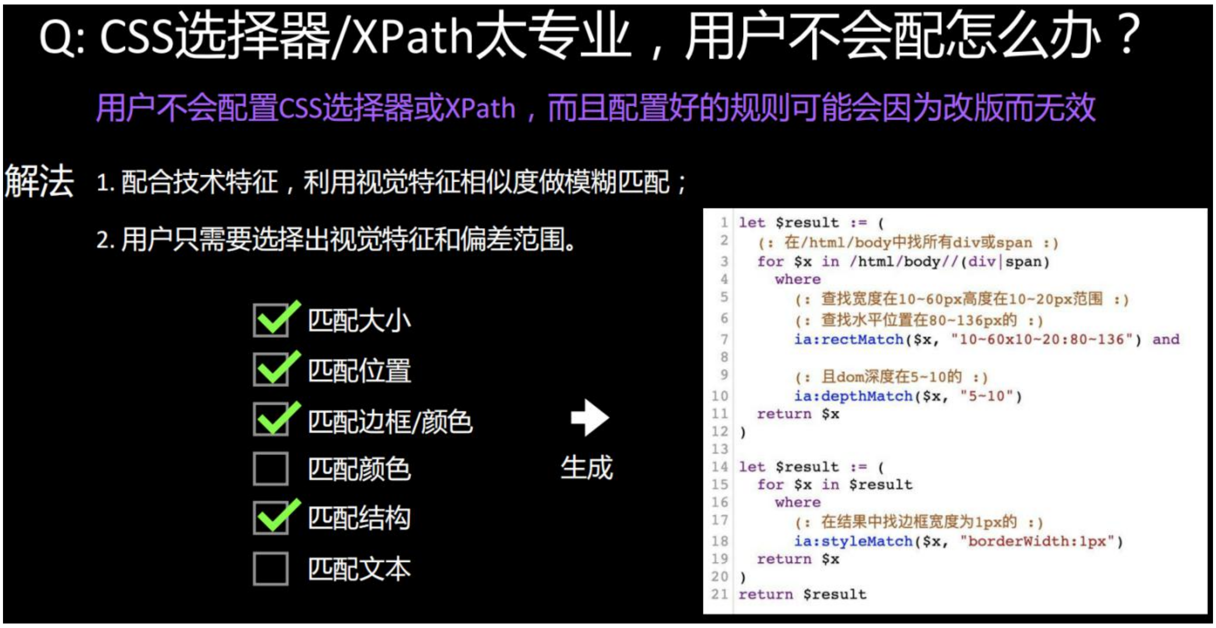
步骤 3：配置交互组件，通过 URL 触发动作



然后，如何将静态的内容变成动态的交互呢？可以通过虚拟协议，即自定义了一个协议的 URL 去触发，如点击带下划线的按钮，出现弹窗等交互动作，都是可以通过 URL 去完成的。

问题一：动态渲染出的页面，可能会因为状态更新导致重新渲染，造成已经标注出来的组件丢失怎么办？

解决方案：监听 DOM 的变化(如 DOM 树结构、DOM 属性的变化)，如果同插入点相关，就是一次关键的变更，重新执行标注，对于已经存在的组件进行更新，否则就创建插入新组件。



问题二：开发人员使用 CSS 选择器/XPath 太专业,运营人员及其用户不会配怎么办?

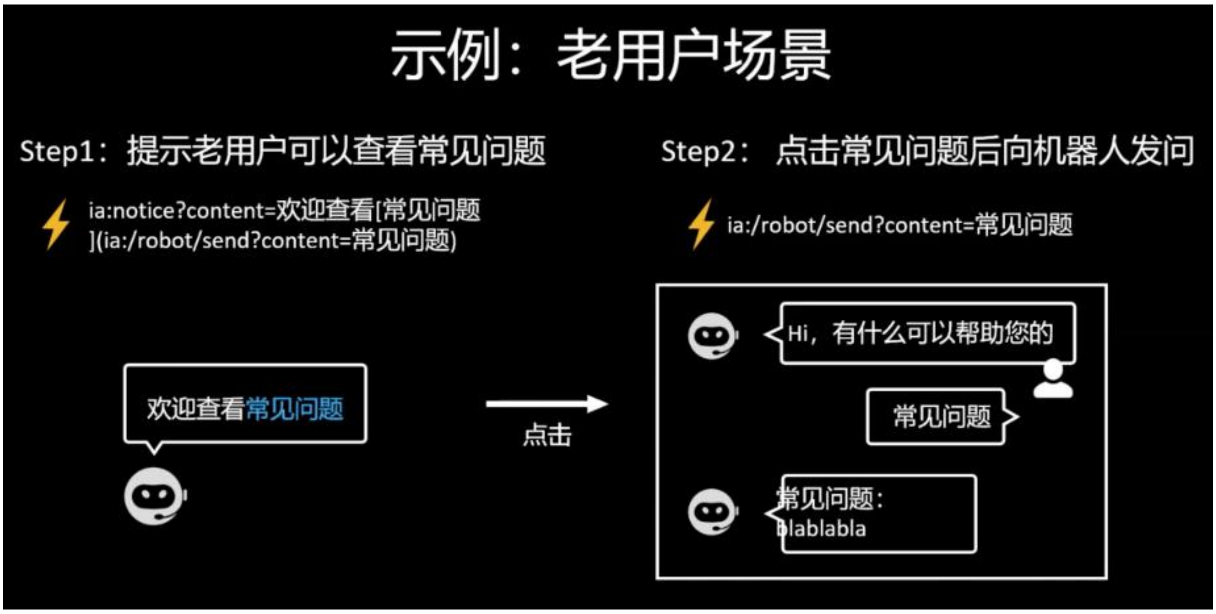
解决方案：实践中是配合技术特征，利用视觉特征相似度做模糊匹配，运营只需要选择出视觉特征和偏差范围。

问题三：如何用 URL 完成比较复杂的交互动作?

对于系统易用性交互而言，是针对中后台系统，技术并不简单，也是需要支持一些复杂交互，配置上去后，不可避免的有会交互动作新交互需求。
我们以下图的场景为例：

- 对新手而言，使用气泡提示新人指南一步步操作；
- 对老用户而言，有什么问题可以帮您？欢迎查看常见问题等等。点击常见问题，会得到一个常见问题列表。





新老用户的场景都是可以通过一个单点的 URL 去完成。而 URL 较为零散，没有组装起来，因此，我们可以条件、顺序、延时等情况来执行。



三、策略编排生成逻辑代码

1、方案核心目标

面对用 ProCode 开发出现复杂逻辑问题时，将所有用 ProCode 开发的交互逻辑，转变为低/无代码配置交互逻辑去解决，并非直接全部替换。

2、方案实施效果

以一例典型高复杂度表单为例，高复杂度的表单中有 3 个场景，所有字段中间的联动关系有 33 个状态，整合了 82 条逻辑规则，开发到 5 个工作日时，新增功能可能会影响到功能开发逻辑，此时转用策略编排，2 个工作日解决，编译出代码 3000 行，效率也提高了。

3、面向复杂交互逻辑的方案简介

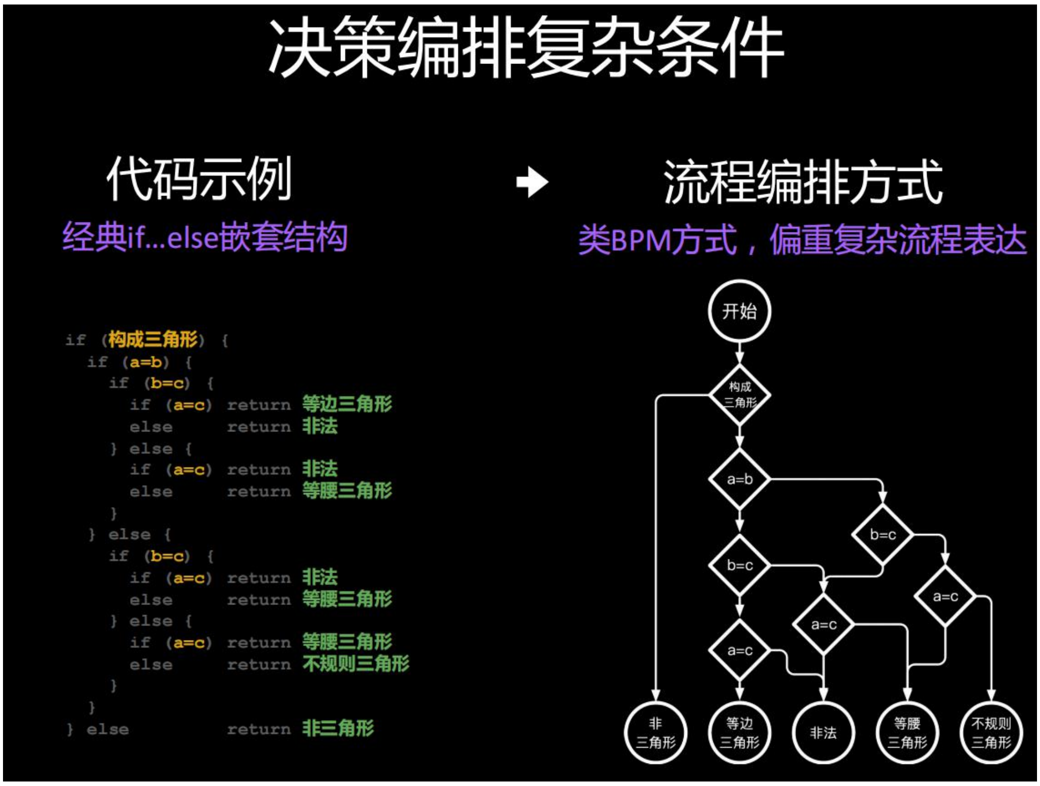
复杂逻辑有较高的验证和研发成本，随着迭代，会成为逻辑黑盒，逻辑和业务都可能存在冲撞，极其容易导致返工，效率极低，怎么办？

- 状态多/难预测/难验证：确定状态来源，明确状态为何改变，可快速验证；
- 联动/条件多：高效的方法论指导，可管理每个状态的联动及条件；
- 技术更迭/代码腐化：由规则直接产生可读性好、具备延续性、技术栈解耦的专业代码。

4、复杂交互逻辑问题

复杂交互逻辑问题转变为解决复杂条件、复杂联动下的状态管理问题。

决策编排复杂条件

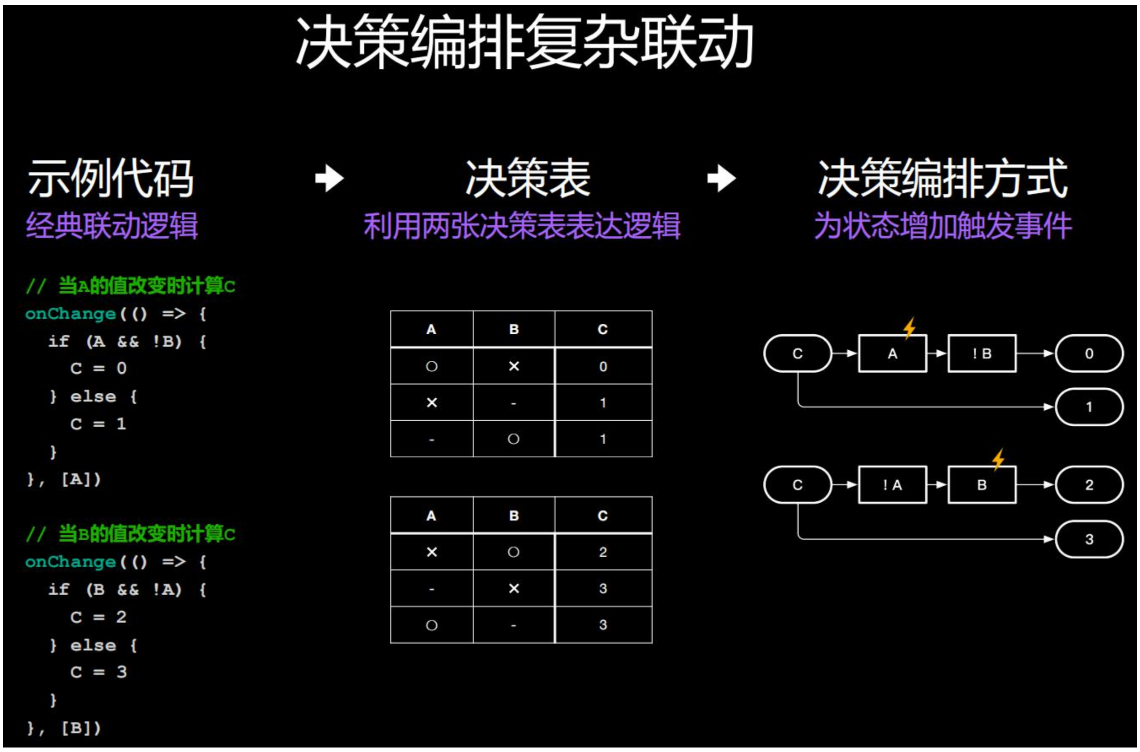


上图左边是一个嵌套结构，判断是什么类型的三角形，右边通过流程编排方式完成了低代码编排，左右结构类似。在理解流程编排结果时，依旧是以编码的思维去理解，偏向复杂流程表达。对于这种复杂流程可能不实用，是否可以换个思维方式，使流程更清晰呢？



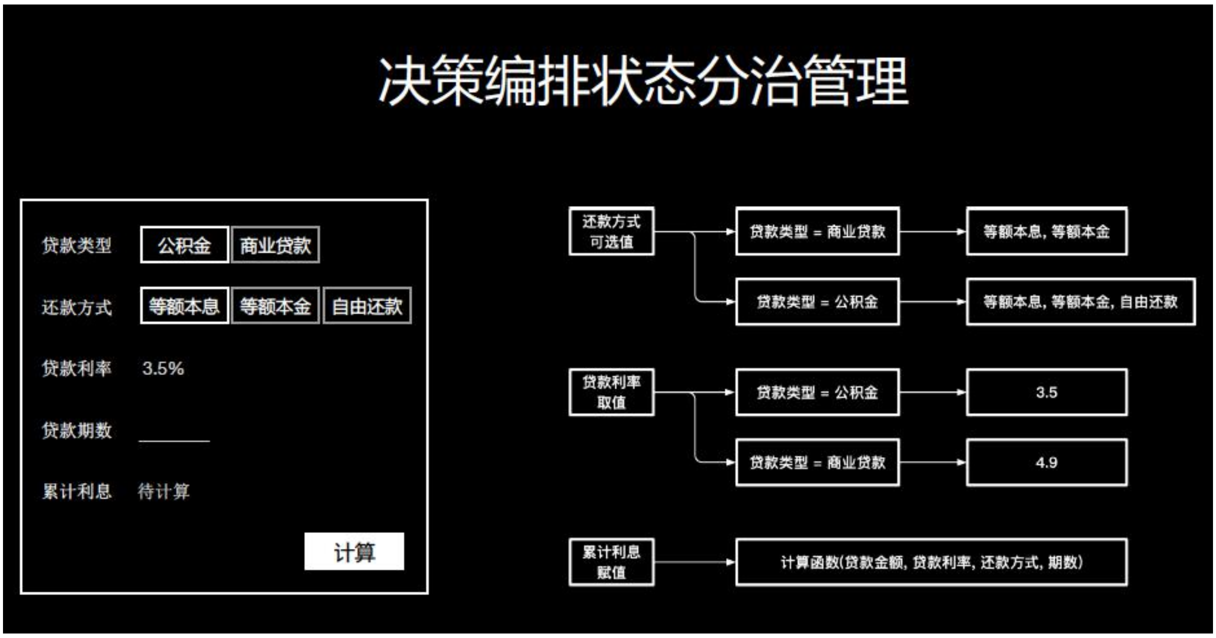
可以使用卫述句式简化条件结构，虽然多出了不少冗余，但是结构却十分清晰，可以轻而易举的转成偏重复杂逻辑的决策表，在业务实践中，经常把复杂的嵌套逻辑转变带有冗余的卫述句式。

决策编排复杂联动



如上图左边是伪代码，当 A 或 B 变化时，分别对应的执行一段逻辑；联动中间的部分转换成决策表，但是决策表里没有联动条件，无法联动，因此，将决策表转换成等价的决策树，在决策树的联动事件加上一个入口点，当 A 或 B 发生变化时，加上一个闪电标志，分别对应的执行一个计算，决策树同时解决了复杂条件和复杂联动问题。

决策编排状态分治管理



以上图贷款计算器为例，贷款类型、贷款方式、贷款利率等指标都是联动的。比如类型是公积金，方式是自由还款，相应的利率会不同，结果可以手动通过计算按钮触发。将此复杂的联动逻辑转变为右边的决策树，逻辑比较清晰。

5、策略编排方案实践

步骤 1：定义要策略编排的对象

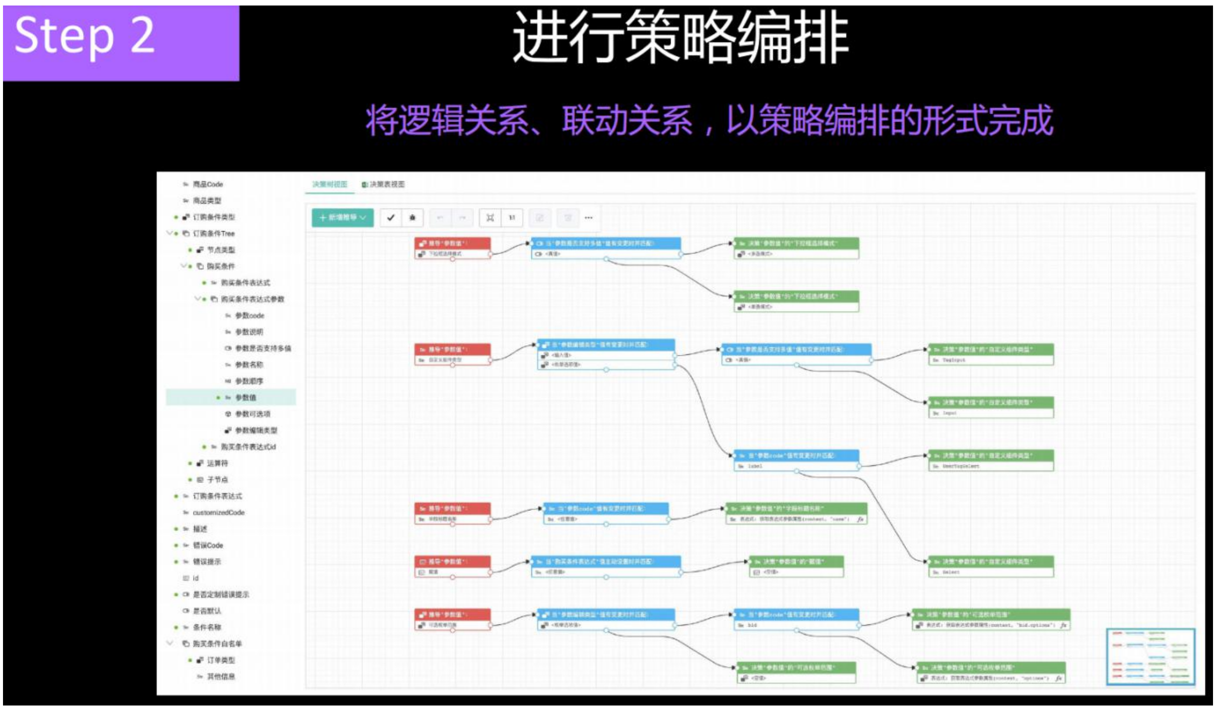
元数据定义了即将策略编排的对象。



实践方案是先确定策略编排的对象，如果是表单，可能策略对象的来源是后端提供的 API 元数据。表单提交的某个字段，可能后端需要接收，此时可以轻而易举的从后端提取出来；又或者是一个自定义表单，元素关系也可以通过自定义编排实现。

步骤 2：进行策略编排

将逻辑关系、联动关系，以策略编排的形式完成。



有了策略编排对象，我们知道了表单或者页面上有多少个字段，字段的状态有可视、只读等，可以将每个状态都作为单个子决策树去编排，就可以将复杂的状态，复杂的逻辑条件关系在决策树里边编排。

步骤 3：调试逻辑及联动

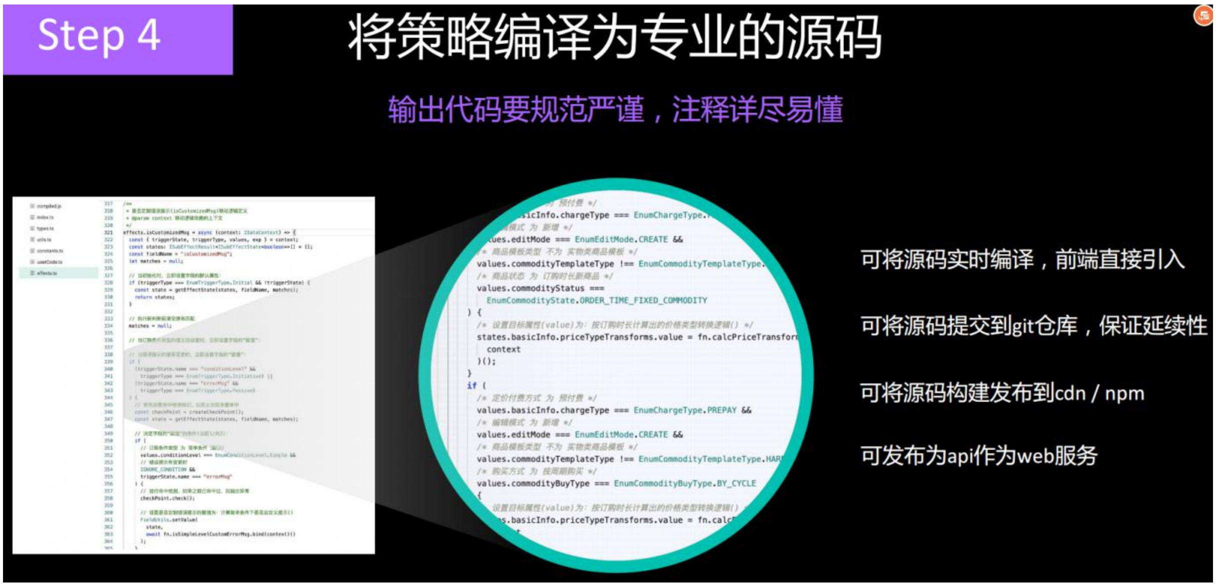
测试驱动开发，保证规则在发布前经过充分验证。



上图左边是将元数据生成模拟表单，供在线验证。上图右边是将策略生产测试用例，标记验证结果。生成的巨大脑图，可以交给非技术非运营者理解业务规则，同时也可以和模拟表达结合在一起做一个验证，看看模拟的表单数状态和实践检验值是否一致。

步骤 4：将策略编译为专业的源码

输出代码要规范严谨，注释详尽易懂。



有了决策树，就可以编译成专业的代码，此代码的特点是：

- 可将源码实时编译，前端直接引入；
- 可将源码提交到 git 仓库,保证延续性；
- 可将源码构建发布到 cdn / npm；
- 可发布为 api 作为 web 服务，增加复用性；

在这个过程中，前端有缓存，在后端没有校验的情况下可能会录入，导致线上出现 bug。如果将这部分逻辑以决策的形式提供给后端，后端用同样的逻辑排查，遇到问题就可以拦截。这里决策树就起到了很大的作用。

四、Next

复杂交互、复杂逻辑具有一定的割裂感，和业务场景密切挂钩。在实践中也是两个平台分别实践。我们下一步要做的就是把复杂交互、复杂逻辑一体化赋能。

希望以上的分享可以给做中后台系统的伙伴们贡献一些思路。

多样化

解读千万级关系网络下的数据探寻可视化技术

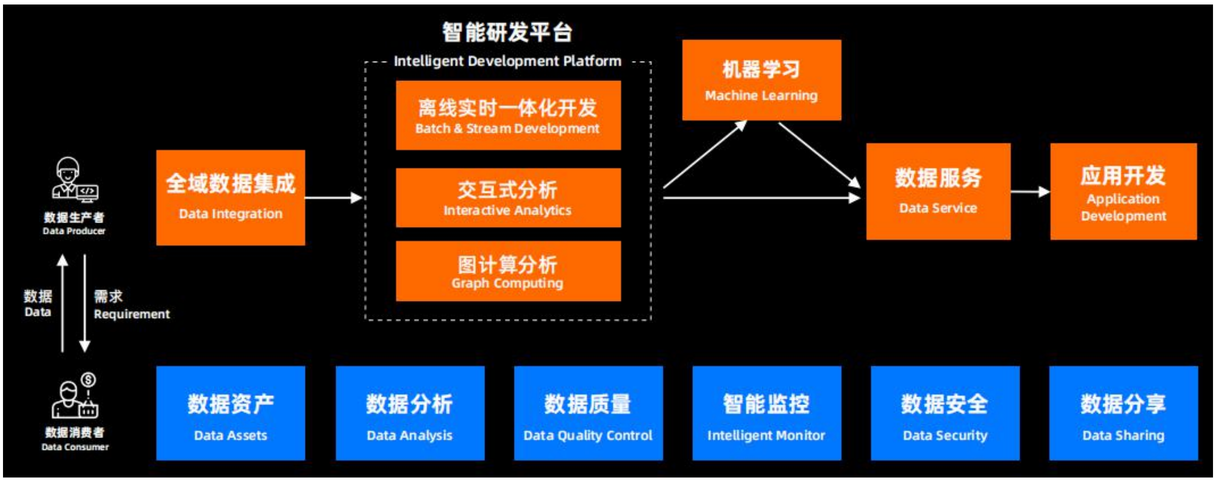
摘要：DataWorks 是一个专业高效、安全可靠的一站式智能数据云研发平台，涵盖数据开发、算法开发、数据服务、数据治理、数据应用等各方面。其核心模块是一个千万级节点的调度引擎。本话题主要探讨了在这样量级的场景下，前端是如何通过图可视分析技术手段帮助用户完成问题发现、链路追踪，问题溯源，血缘分析等。

分享人：刘东奇(佛肚)，阿里巴巴可视化小组图可视化方向负责人，AntV 团队成员，DataWorks 产品可视化负责人。 曾独立完成 Graph Studio 图数据管理、图可视分析平台构建。Graphin 核心参与者，《AntV 图可视分析解决方案白皮书》核心参与者。 目前主要专注图可视分析技术产品化以及多布局融合技术场景化落地。

一、DataWorks：全域智能大数据平台

阿里云飞天大数据平台是阿里巴巴 10 年大数据建设最佳实践的结晶，每天有数万名数据和算法工程师正在使用飞天大数据平台，承载了阿里巴巴 99%的数据业务构建，同时广泛应用于城市大脑、数字政府、电力、金融、新零售、智能制造、智慧农业等各个领域的大数据建设。

DataWorks. 作为整个飞天大数据的重要组成部分，它基于 MaxCompute/EMR/ MC-Hologres 等大数据计算引擎，为客户提供专业高效、安全可靠的一站式大数据开发与治理平台。



二、图可视化技术

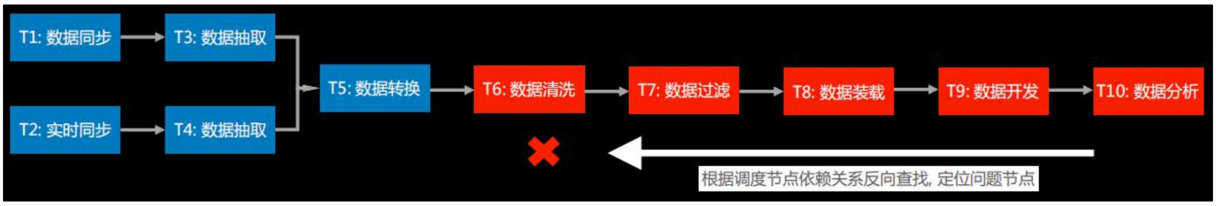
从前端角度看，可视化是数据与数据之间关系的最佳呈现方式。目前，图可视化技术被广泛运用于阿里巴巴产品中，以解决一些实际场景问题，如：运用图可视化技术呈现血缘关系、编排业务，节点调度方面运用图可视化技术提供更多分析能力等等。



DataWorks 中的图分析场景

1、DAGRE 布局+视觉通道映射（颜色）技术绘制节点依赖关系

在拥有庞大数据量的场景下，需要引入大数据体系来支持数据计算和分析。通用大数据加工流程（如下图），将存在于数据库或系统日志中的信息通过大数据同步工具加载到数据引擎中，流经数据抽取、数据转换、数据清洗、数据过滤、数据装载、数据开发到数据分析，最后得到所需数据。



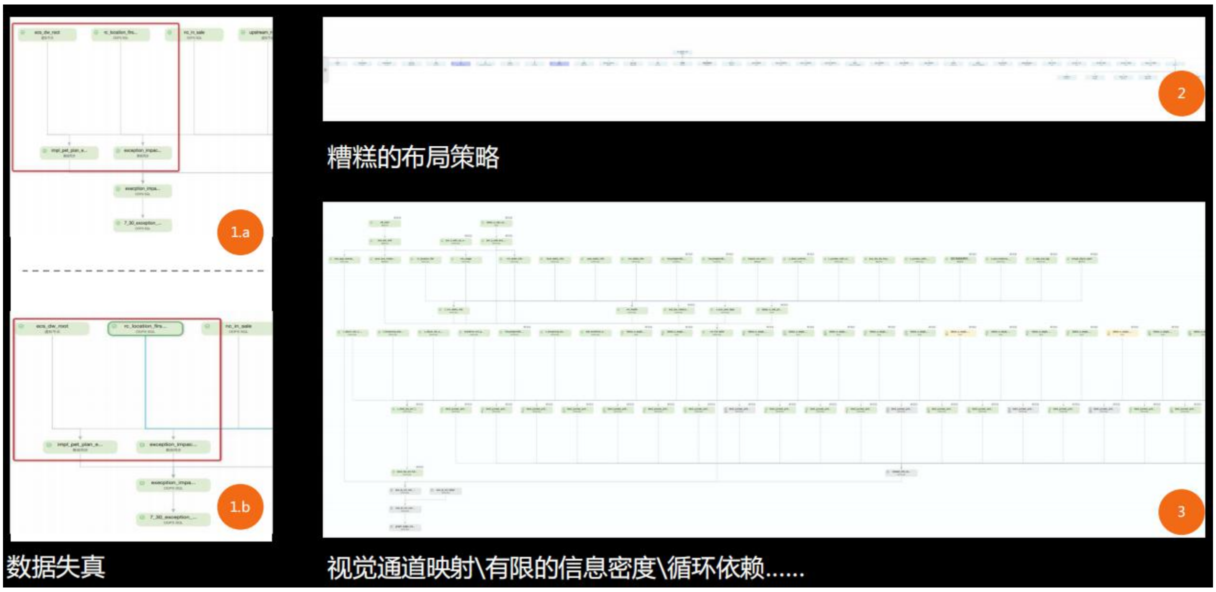
通用大数据加工流程

这一流程复杂且环环相扣，而 DataWorks 的核心旨在确保其稳定的被调度和被执行。但这其中的大部分节点是用户根据其业务场景和需求开发，因此不可避免的会出现一些问题，这些问题对下游会产生很大的影响。

比如：T6 数据清洗出问题，就会导致下游 T7-T10 所有链路堵塞或未执行。当 T10 节点数据无产出，需要快速定位失败节点，可以通过使用 DAGRE 布局+视觉通道映射（颜色）技术绘制节点依赖关系，根据调度节点依赖关系反向查找，定位有问题节点。

2、真实场景中的挑战

然而在真实场景中，使用 DAGRE 布局+视觉通道映射（颜色）技术仍然面临极大的挑战（如下图）：



- 数据失真：图中 1.a 所示画布节点依赖关系；1.b 所示真实数据无节点无任何依赖；
- 糟糕的布局策略：下游节点过多，无法一屏展示；
- 视觉通道映射\有限的信息密度\循环依赖：节点区分度不高、节点数过多、关系复杂溯源难度高；

以上是一些看得到的挑战，还有一些看不到的挑战：

- 数据庞大：节点数超过千万；
- 同层宽度：同层节点数多，目前最多十万+；
- 关系复杂：依赖复杂，跨周期、小时依赖等；

针对以上挑战，尝试了一些解决方案：

1）从渲染技术出发，提升节点渲染数量

- 离线渲染技术

服务端渲染（PlanUml）；

Offline Cavanas；

- 局部渲染技术

追求极致性能的@antv/G6 引擎：

支持局部渲染技术，只渲染可视窗口内信息；

使用脏矩阵渲染，在每一帧中进行最小区域的重绘；

使用 R-tree 空间索引加速区域查询；

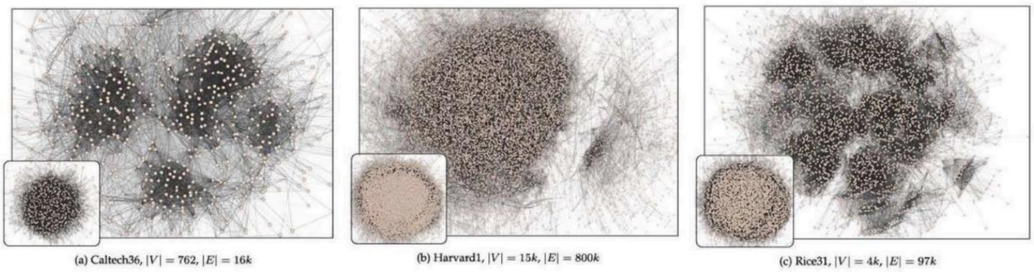
- 双引擎切换能力

Neo4J Bloom 的双引擎机制：

全局视角采用 GPU Powered Rendering 能力，展示更多数据；

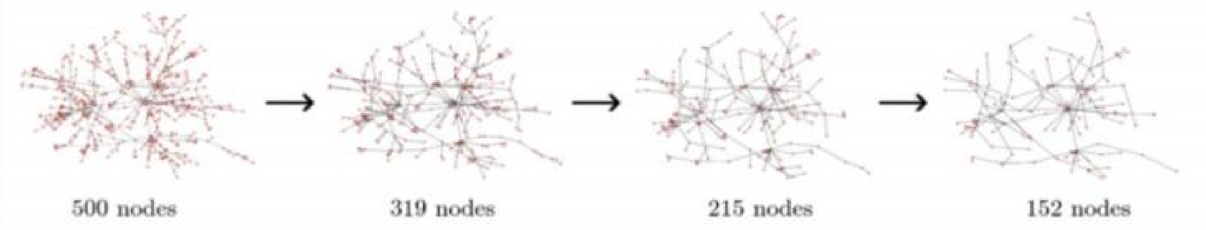
局部视角采用 Canvas Rendering 能力，展现更丰富的细节；

通过渲染技术的改造，目前可以实现万级节点渲染能力和千级交互。但过度追求渲染能力的会导致“毛团”现象，使关系网络可读性下降。

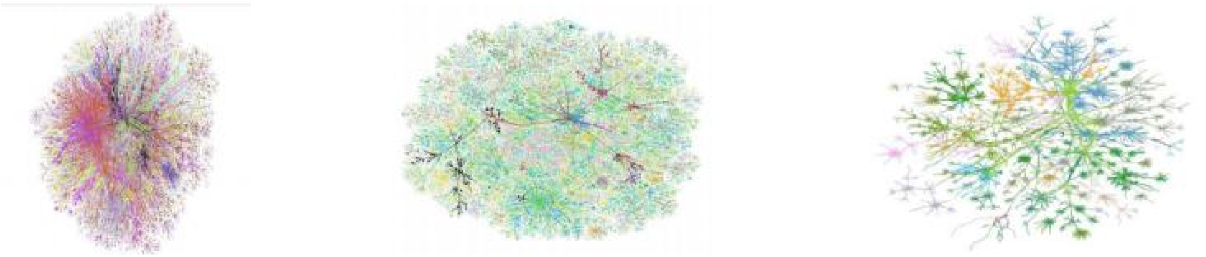


2）针对毛团现象的解决方案：图简化技术

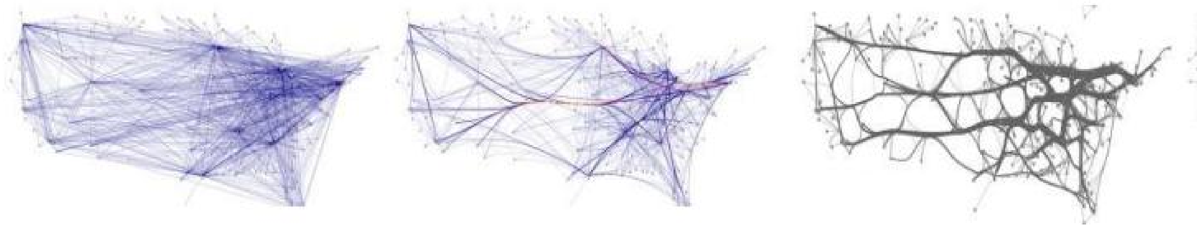
点聚合（Node Aggregation）：目的是提取出大规模数据不同“粒度”的图，前端首先展示最粗粒度的图，终端用户通过放大、下钻的方式查看更细粒度的图/子图；



边剪枝（edge bundling）：通过反向删除算法、Prim 算法构建最小生成树，通过只保留关键路径减少视觉；



边绑定（Edge Bundling）：边缘捆绑技术被设计为在视觉上将相似的边缘捆绑在一起，以减少图中的视觉杂乱。这种捆绑可以帮助突出重要的边缘模式，并且常常使人更容易发现有趣的连接或重要的数据。

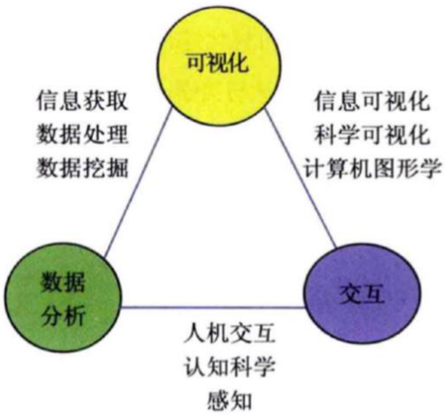


三、可视分析

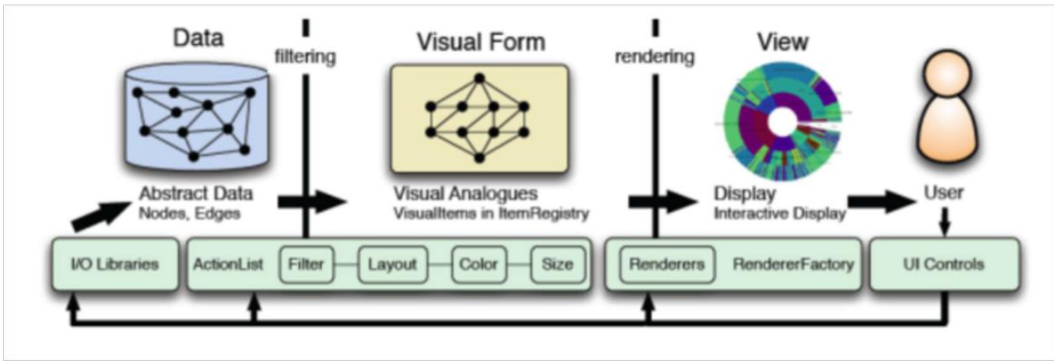
1、可视化 vs 可视分析

点聚合方式与图可视化方式的区别在于，点聚合不再强调信息的展示，而更多的是人的参与和人机交互，这在可视化领域中被称为可视分析。对比传统可视化与可视分析会发现：

- 传统的可视化：大多立足于先验知识，解决一些具体且可预期的任务时有一定的优势；
- 可视分析：将“人”这个独特因素融入进了数据分析的流程中。通过人机交互技术，将更多的重点放在了人的意会与推理上，让人在分析任务中参与了主要的分析与决策过程。有了可视分析的支持，能够在很多单纯依赖算法分析无法解决问题的场景中，进行可视知识发现，获取有价值的信息。



可视分析组成



数据可视化流程

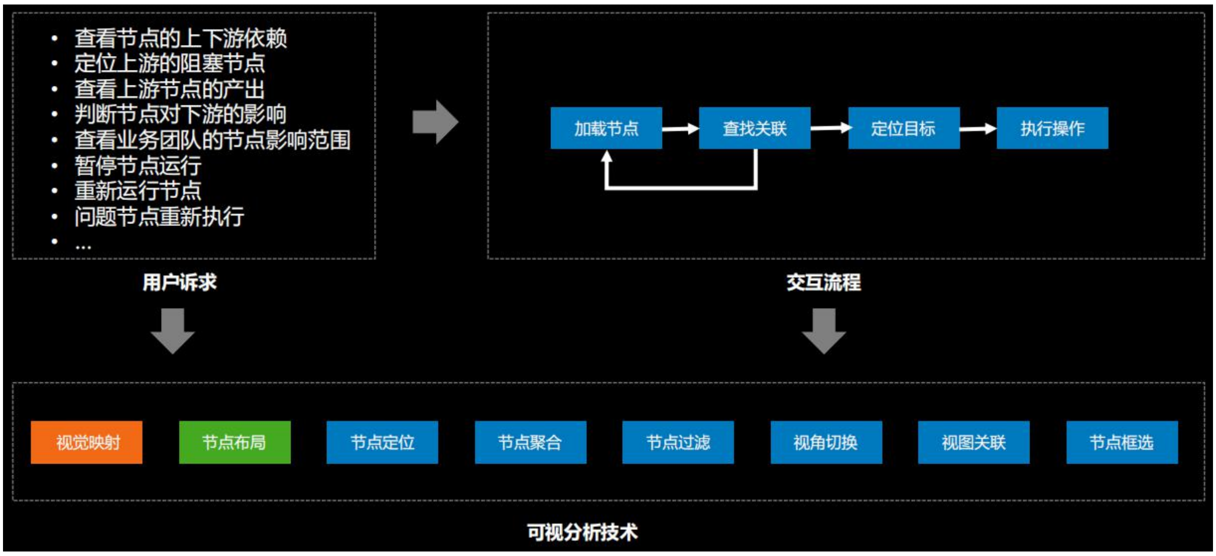
2、从用户需求看产品交互

用户诉求包括：

- 查看节点的上下游依赖；
- 定位上游的阻塞节点；
- 查看上游节点的产出；
- 判断节点对下游的影响；
- 查看业务团队的节点影响范围；
- 暂停节点运行；
- 重新运行节点；
- 问题节点重新执行；
- ……

然而对于前端，无论用户诉求的内容是什么，其交互逻辑都是一致的（如下图），即加载节点、查找关联，并对信息做反复查找和分析，最终确定目标并执行操作。

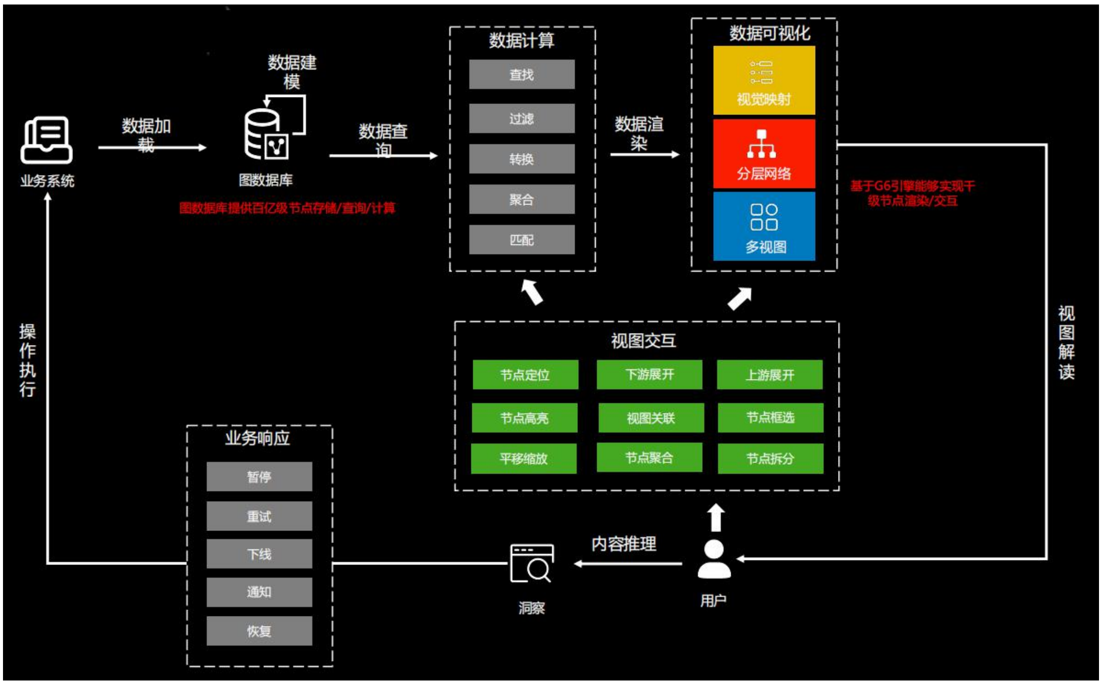
前端进行可视分析会关注的技术包括：视觉映射、节点布局、节点定位、节点聚合、节点过滤、视角切换、视图关联、节点框选。



3、可视分析技术架构

如下图所示，业务系统承担千万级节点任务调度，首先将信息载入图数据库中，图数据库与传统关系型数据库的不同在于，它更关注关联查询计算的效率，可以提供百亿级节点存储、查询和计算。

基于这样的计算引擎封装了一套计算逻辑，以实现从千万级节点中加载出需要分析的数据，通过这一计算过程可以将数据降级到百级别甚至更小量级让数据更具分析性。加载出的数据可以使用数据可视化技术（如：视觉映射、分层网络、多视图）进行数据渲染，用户可以基于渲染出的视图进行视图解读，然后运用视图交互的能力去查询或实现，最后通过反复沟通过程进行内容推理得出结论，基于这个结论执行相应的业务响应。

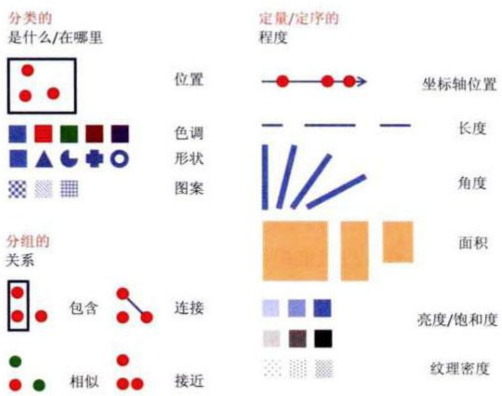


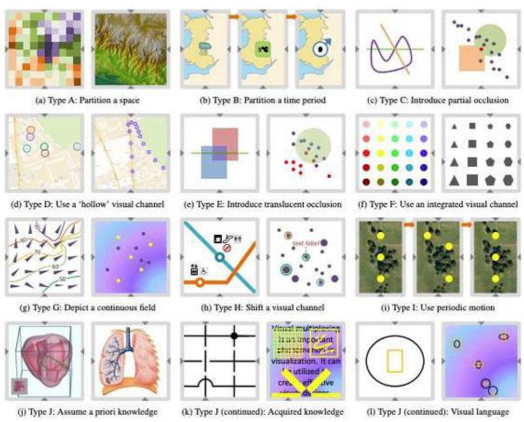
可视分析技术架构

4、视觉映射

视觉映射是数据可视化技术的重要组成。提到视觉映射就会联系到视觉通道，它是可视化编码的一部分，用来控制可视化编码标记的展示特征，包括标记的位置、大小、形状、方向、色调、饱和度、亮度等。

下图左边是有限的视觉通道，以及表现力排序；右边是通过视觉多通技术实现多可视化信息堆叠。





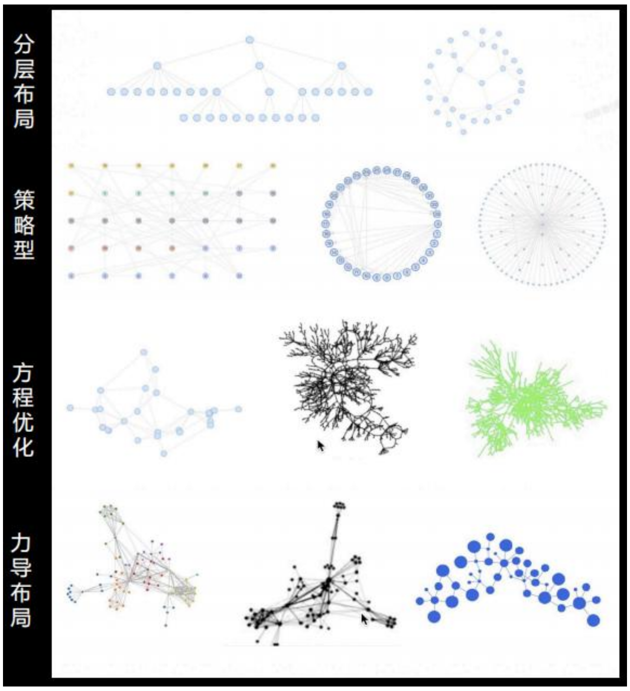
在使用视觉多通技术场景下，选择“位置、色调、亮度/饱和度、包含、关联”再配合空间分割和多通道整合实现视图映射。



5、布局

通用布局策略局限性

图可视化分层布局策略包括：分层布局，策略型布局，方程优化布局、力导布局等。



图可视化常见布局策略

在实践中要根据业务场景和分析目的有针对性的选择布局模式。

业务场景	分析目的	技术方案
套现网络/闭环网络	看清楚是否存在环	•圆形布局 circle •力导布局 force
分层网络/流程网络	看清楚层次	•有向分层 dagre •树形力导 force-tree
无序网络数据	看清楚 点和边，不要点线交叉	•力导布局 force •fruchterman
暴涨网络 / 坍缩网络	关注中心位置的情况	•同心圆布局 cencentric •径向布局 radial
团伙聚集网络	关系聚集，分类现象	•同心圆布局 cencentric •聚类布局 node combo •Circle packing
多边有向网络	看清楚 边的方向	•弦图布局 arc

布局的核心：

- 清晰展示节点信息；
- 清晰表示依赖关系；
- 帮助用户快速分析问题；

然而在真实场景下的布局依然会存在很多问题，如基于 Sankey 布局构建的分层网络的问题：

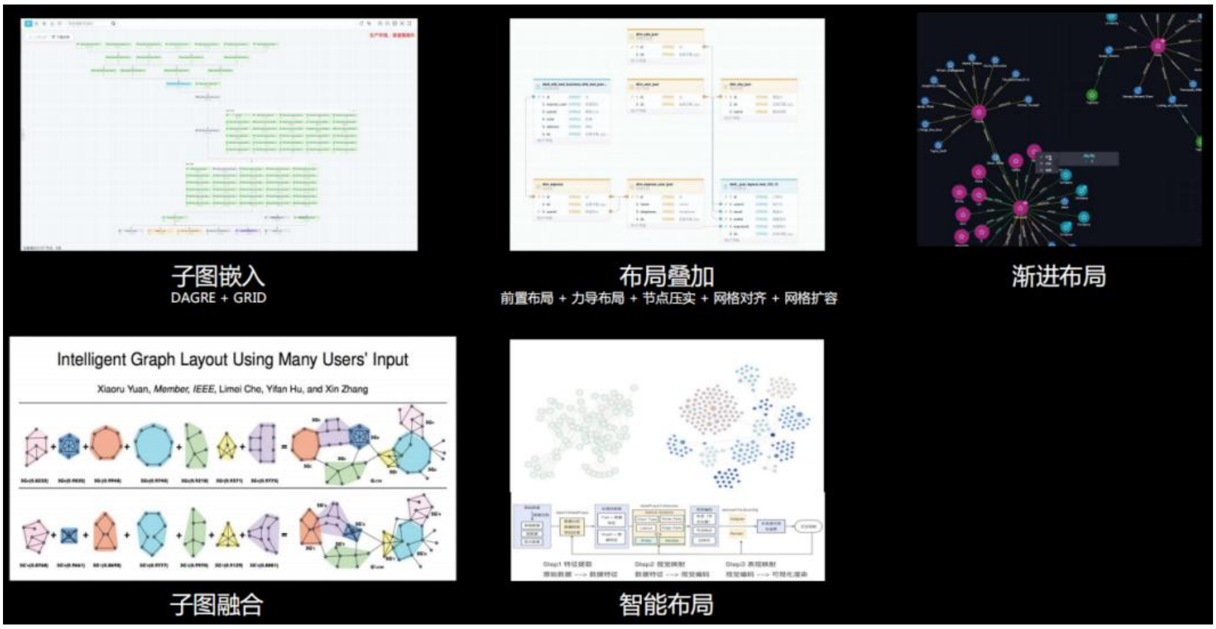
- 图宽特征下造成布局浪费；
- Sankey 布局缺陷，布局中不考虑边交叉问题；
- 线的路由算法，合并线后丢失方向信息。

布局融合技术

布局融合技术是针对以上问题的探索，分为：

- 子图嵌入布局 DAGRE+GRID：将 GRID 布局嵌入至分层布局网络从而解决图宽的场景；
- 布局叠加：前置布局 + 力导布局 + 节点压实 + 网格对齐 + 网格扩容；

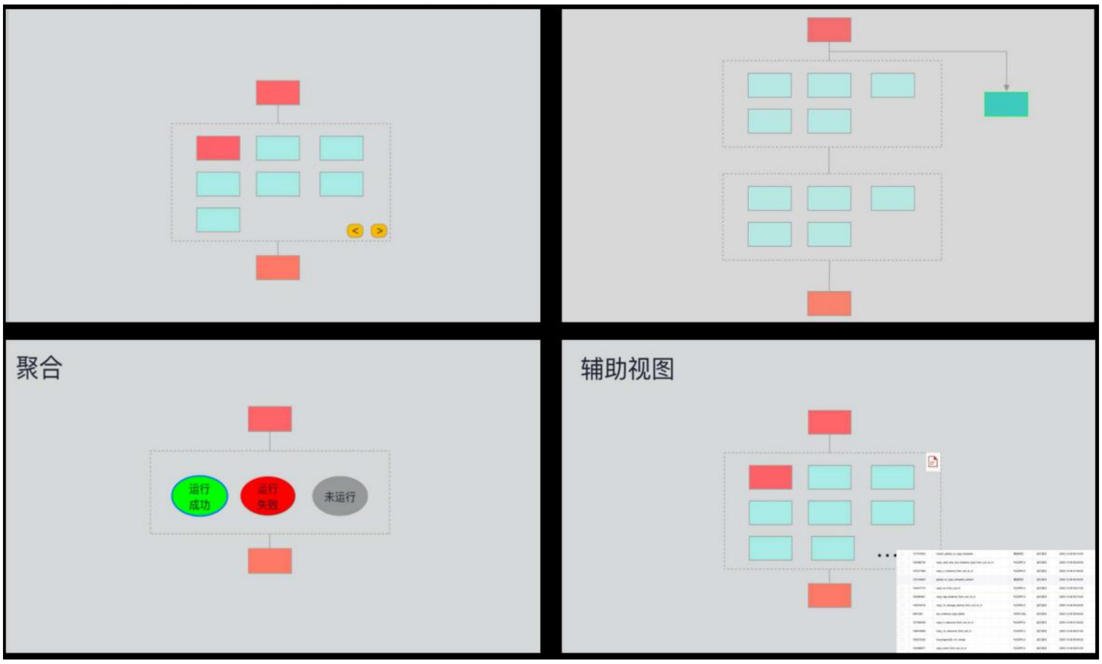
- 渐进布局；
- 子图融合；
- 智能布局；



通过分组设计解决位宽问题

分组设计：

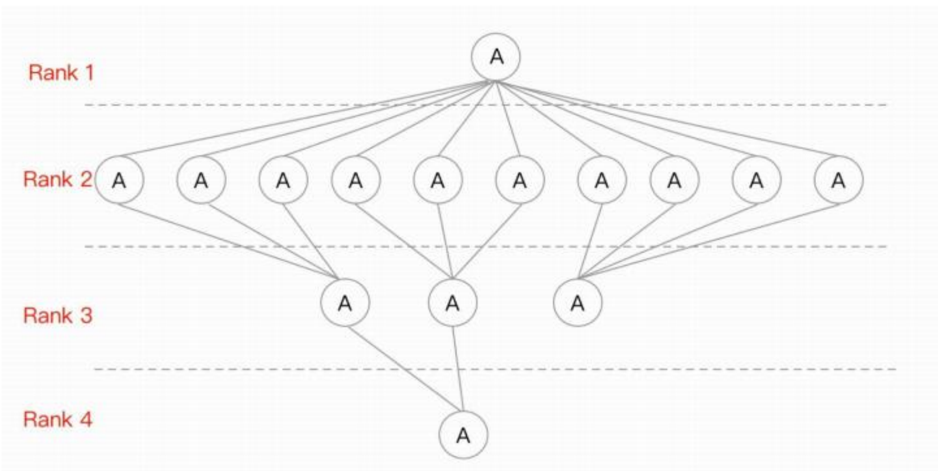
- 同层节点置于一个分组中，支持换行翻页；
- 提供聚合工具，帮助用户快速分析；
- 通过辅助视图，展示全量节点；



增加分组策略后的布局算法

分层网络+分组设计：

- 节点分层：根据依赖关系进行分层，采用紧致树分层算法。
- 以下图为例，根据算法分成四层



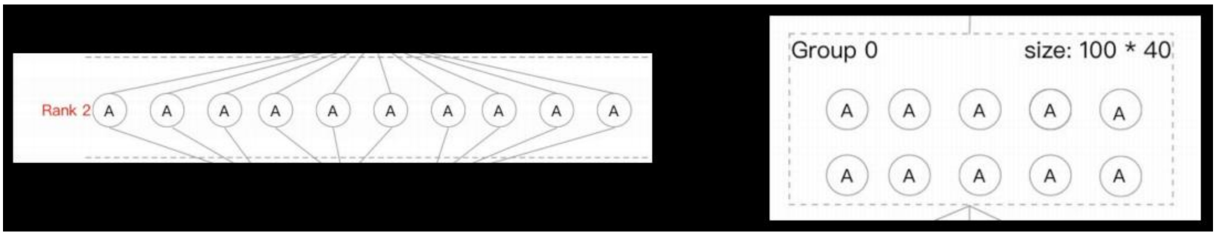
- 虚拟分组：判断每层节点数量是否大于 MAX_LEVEL，是则生成虚拟分组节点。

根据图中节点的数量计算虚拟分组的大小以及其中的节点要如何排列；

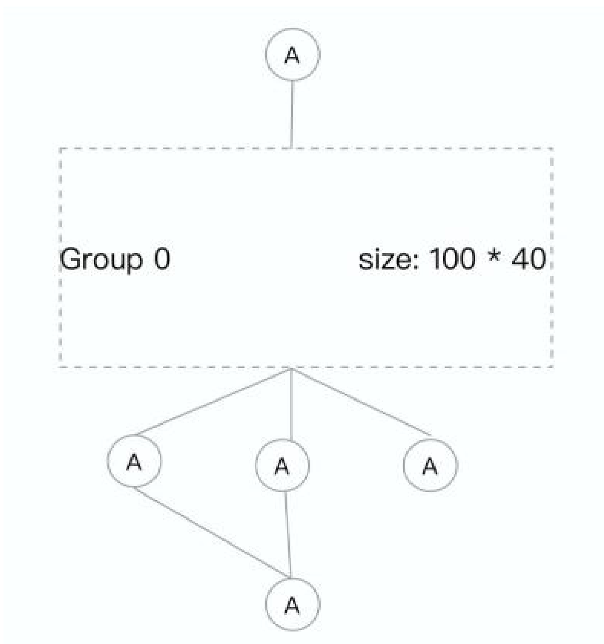
计算方法：

Width=(NodeWidth+NodeGap)xColumn+2xGroupPadding

Height=(NodeHeight+NodeGap)x(NodeNum÷Colum)+2xGroupPadding



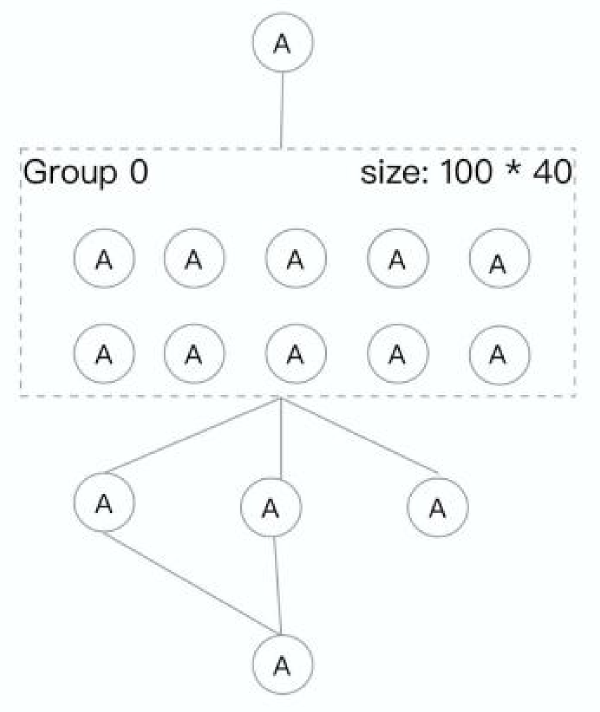
- 节点布局：虚拟节点以及正常节点进行 Dagre 布局；



- 分组内嵌：根据虚拟节点的坐标，转换层内节点的相对坐标，放入虚拟 Group 中；

NodeX=InnerX+GroupX

NodeY=InnerY+GroupY



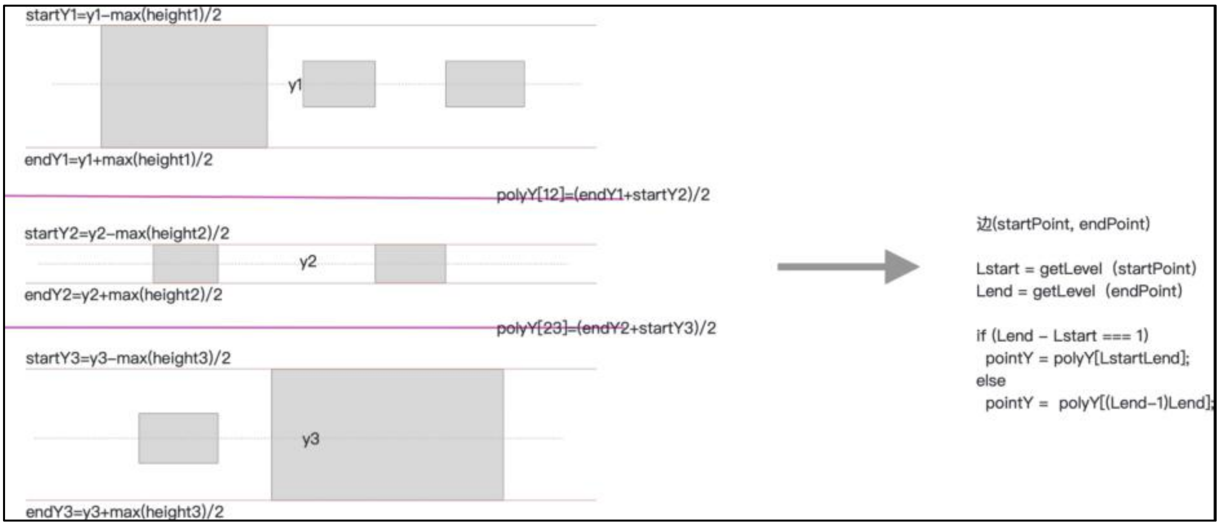
6、布线策略优化

除了节点布局，另一个需要考虑的是布线。针对布线策略做了以下优化，以提升可读性：

- 将 SanKey 布局替换为 Dagre 布局，尽量避免边交叉；
- 线的路由算法，拐点路由计算；
- 直角改为圆角，增加方向性区分；

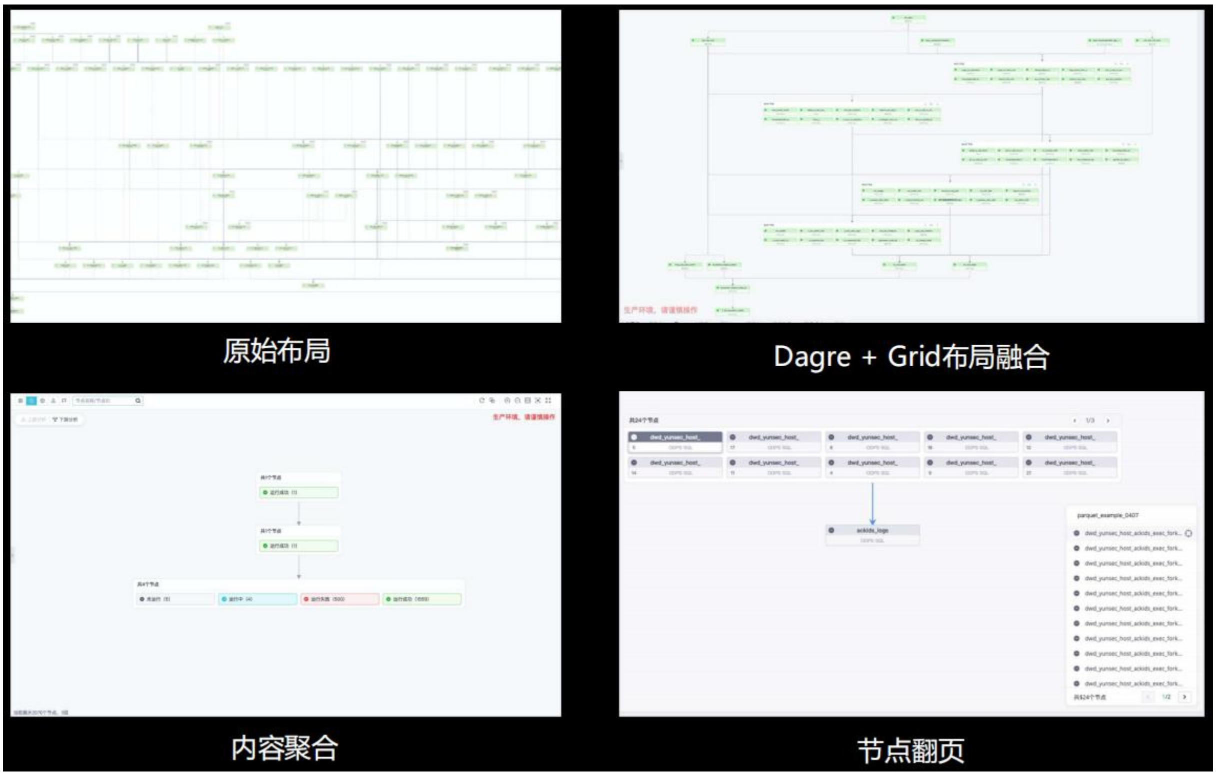
实现步骤：

- 层级分割：根据所有外部节点（去除内部节点）的 y 坐标，进行分组，划分层级；
- 边界计算：根据该层级所有节点的 y 坐标已经高度，计算出层级区域的界限；
- 拐点确认：根据层级界限，以及设置的 gap，计算出层级之间拐点的 Y 坐标；
- 绘制曲线：所有曲线的拐点，按照曲线所处层级的拐点 Y 坐标绘制。如果是跨多个层级的边，按照最后一个层级的拐点绘制（原因：拐点离终点越近，越容易跟踪线的来源）



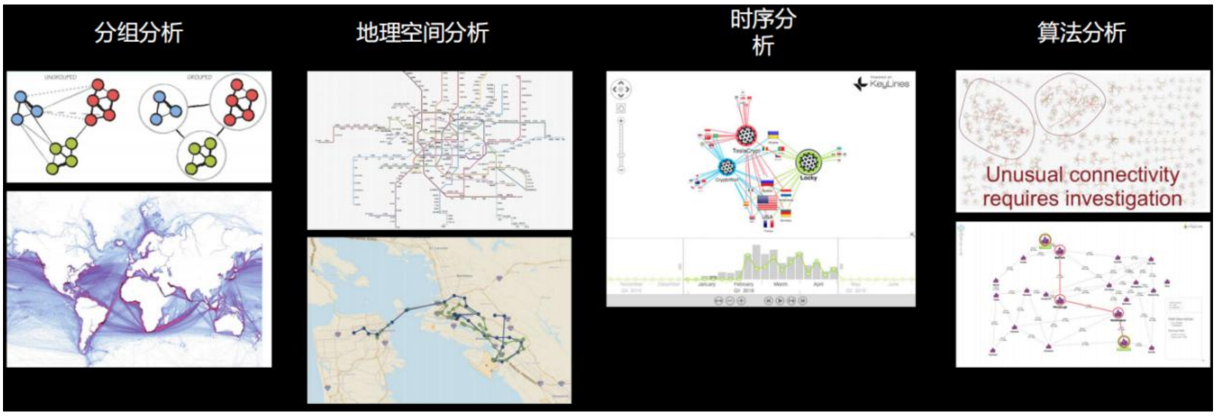
布局效果对比

通过对比可以看到，从最初的原始布局的不可读状态经过 Dagre+Grid 布局融合，配合内容聚合和节点翻页，逐渐转化为具可读性和可分析性的图。



7、分析方法

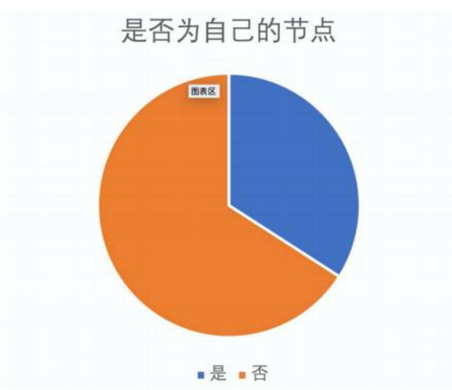
除了上述的视觉映射和通道布局，图可视分析的另一个主要部分是分析方法，包括：分组分析、地理空间分析、时序分析、算法分析，根据场景不同运用不同的分析方法。



8、场景化分析能力

背景：

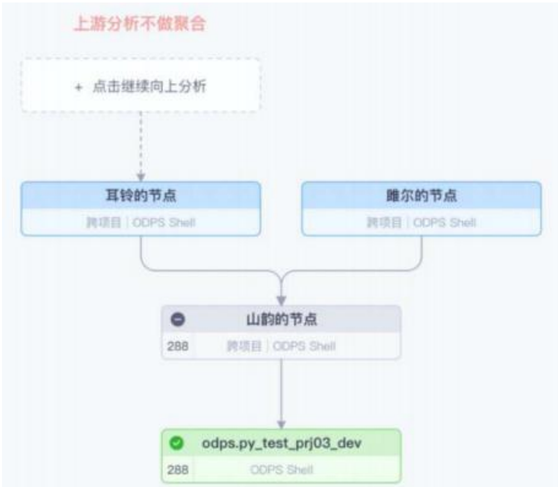
- 用户在什么情况下会点击 dag 的展开上下游，对应用户想要解决的问题是什么？
- 业务猜想验证：
- 展开下游：用户会针对“自己 owner”进行展开下游操作，以判断节点对下游的影响；
- 展开上游：用户会针对“错误/未运行”节点进行展开上游操作，去定位上游的阻塞节点。





根据以上验证结果得出两种分析方法：

- 上游链路分析：针对未运行实例展示阻塞链路；

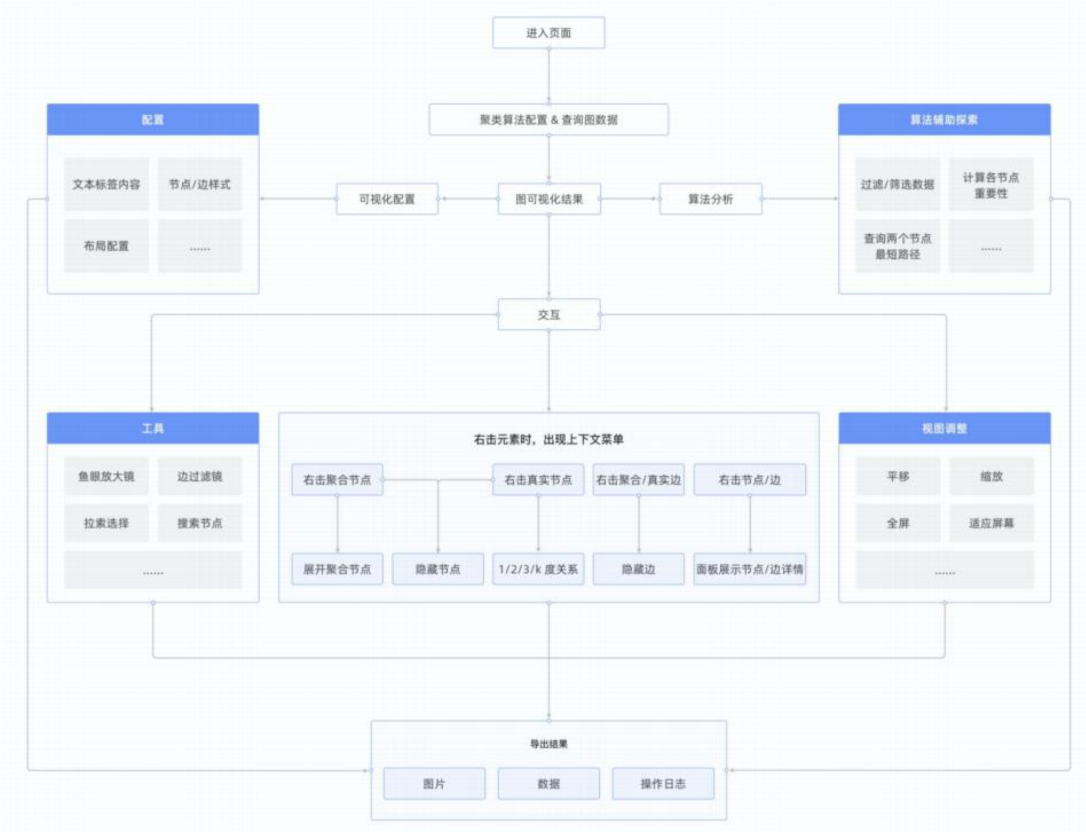


- 下游影响分析：查看节点的下游影响面，提供责任人、项目等多维度视角。



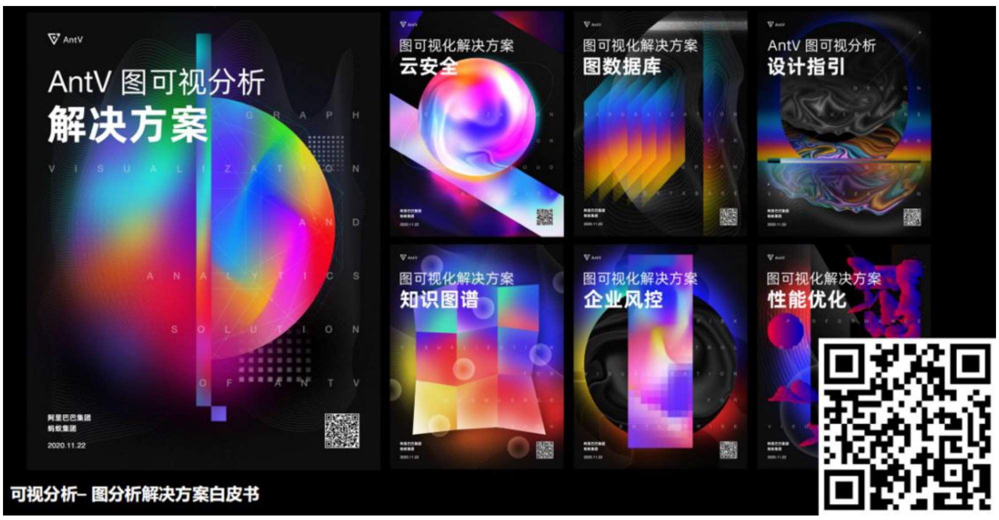
9、完整分析链路

经过以上的改造，抽象出完整分析链路图（如下图）。首先加载数据到界面中，基于界面的预配置和信息将数据进行视觉通道映射，接着进行三个层面的交互能力：工具层、视图调整、以及中间的场景化分析，用户经过反复沟通后推断出结论，最后通过图片、数据、日志的方式呈现。



四、图分析解决方案白皮书

以上是通过示例介绍可视分析技术在 DataWorks 大数据的场景下的运用，更多图可视化分析解决方案现已生成白皮书，感兴趣的朋友可以通过下图二维码获取。目前我们也在探索将这些解决方案的能力通过工具化平台化的方式提供给开源用户，希望在不久的将来可以实现。



向量到墨水:从绘画视角解析三维模型的漫画风格渲染技术

摘要：使用三维技术高效制作动画、游戏，已成近年来的趋势，而同时一大批主打动漫风格的作品仍然需要通过灵动的二维画风进行视觉呈现，如何将三维模型赋予观众熟悉和热爱的动漫风格便成为了一个非常有趣的技术挑战。我们将从绘画视角出发，结合技法、艺术史解析独特的 Cel 漫画渲染技术，带给你一次人文艺术与技术碰撞出的独特体验。

分享人：黄河清(改青)，2014 年本科毕业于北京航空航天大学新媒体艺术与设计学院，插画专业。期间在幸星动画实习，任平面动画师、背景画师；后于时尚集团-国家地理杂志实习，任杂志内容美术编辑。2014 年进入美国纽约大学 Tisch 艺术学院获得人机交互硕士。2016 年纽约大学 Tisch 艺术学院毕业，并担任 Interactive Telecommunication Programme 驻院研究员。2017 年加入美国 Ubiquiti-Networks 公司，主导开发基于 Three.js、Unity 引擎的三维交互的智能企业级网络设备开发、机房运维工具等。2020 年任纽约大学 Tisch 艺术学院春季学期学期的客座教授。2021 年加入阿里巴巴云智能-计算平台事业部前端团队，主要从事常规前端开发以及三维展示方案。

渲染一词源于中国画的传统技法，在计算机的语境中，是指通过显卡以及图形算法计算图形特征并输出到屏幕的过程。那么把三维渲染成二维的动漫风格是如何完成的呢?

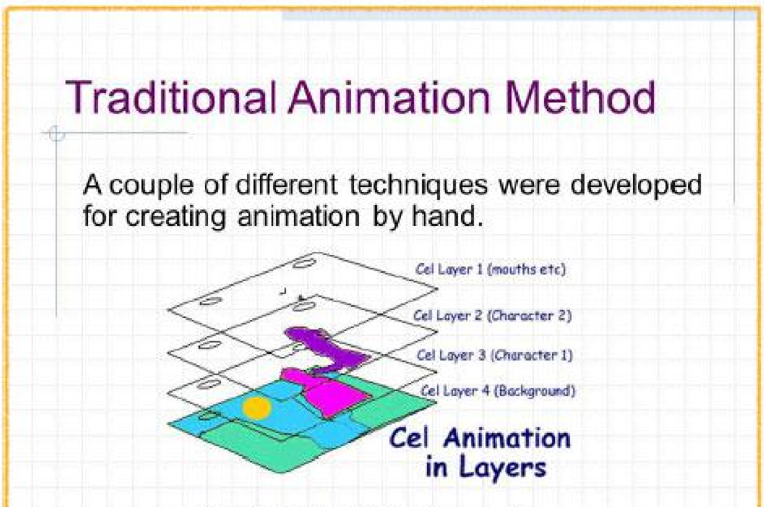
一、动漫风格的绘画特点

1、CEL 动漫风格

通过三维建模软件建模，再用特殊渲染器渲染出的作品，在技术上统称为 CEL 动漫风格。



CEL 源于传统 2D 动画制作流程中的透明图层（Celluloid），2D 动画是将前中后景分别画在不同透明图层上，这种制作方式使动画制作效率达到最高。



胶片图层 Celluloid Film Composition


Source: Detective Konan, TMS Entertainment


Source: Dragon Ball, Toei Animation

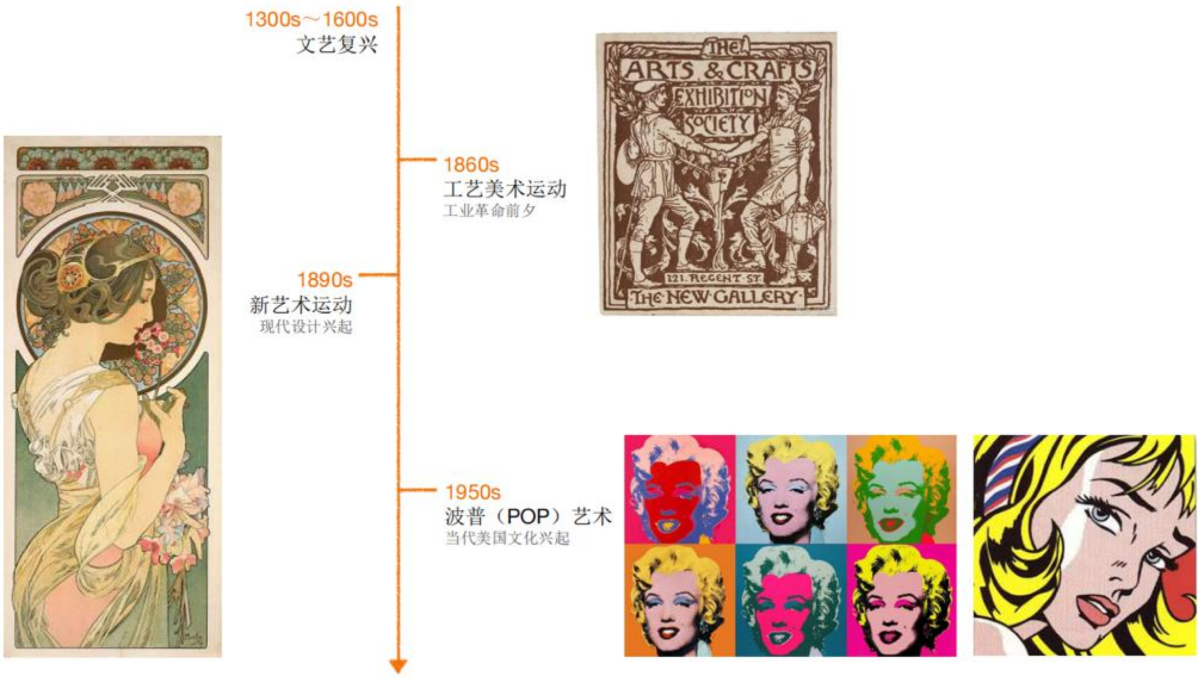
随着 3D 技术的不断进步以及 3D 制作成本降低，越来越多的公司选择直接运用 3D 进行游戏的构建和开发。然而更多的游戏用户仍然对 2D 的 CEL 风格情有独钟，于是将 3D 渲染成 2D 的渲染技术就应运而生。



示例：用 3D 开发以 2D 呈现的游戏

2、艺术的世俗化

从工艺美术运用以来，艺术可以更好的人类生活服务，1950s 波普艺术的兴起奠定了日后漫画艺术的发展。



任何漫画风格都是基于线条为主、用颜色点缀的构成方式。



Clamp



Hikkikomori



DC



成稿



线



色块

二、绘画语言转变到渲染的挑战

动漫渲染的两大难题是线和色块。

1、线的形成

线并不是客观存在的，而是大脑对实物的提炼。通过线表现，是高级的绘画方式，它不能是机械的，要通过穿插、粗细、虚实变化，来体现空间感。漫画风格的线渲染难度在于，不仅要渲染出不存在的东西，而且还要渲染得有灵魂。

线出现在哪些地方？

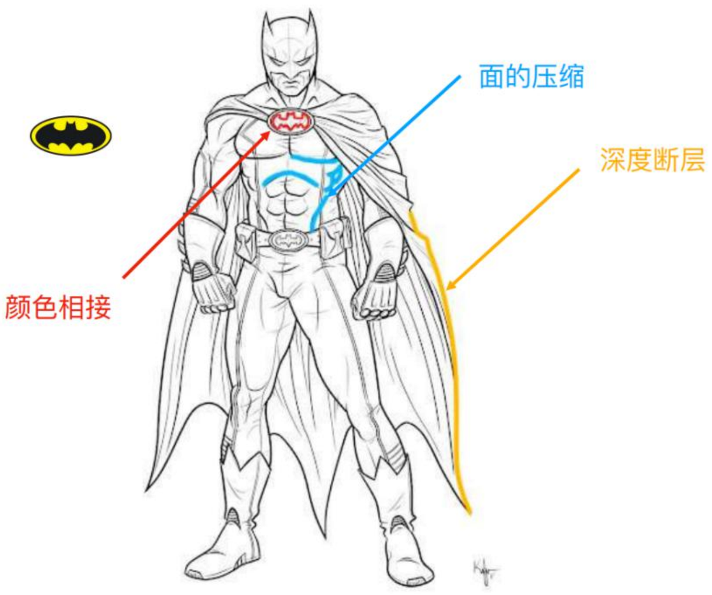


线形成的三个原则：

- 颜色相接：当不同的颜色相接的地方出现边缘线；
- 深度断层：当物体存在前后关系时，前面挡住后面的，深度的变化形成线；
- 面的压缩：平面在旋转中会压缩成线；

以上三个原则相互补充构成我们看到的“线”。再以下图为例：

- 颜色相接：纹理特征
- 深度断层：外边缘
- 面的压缩：内部细节线



2、量化判断条件

- 颜色相接：由色相(Hue)、对比度(Contrast)、饱和度(Saturation)差值产生；
- 深度断层：由深度(Depth)变化的不连续性产生；
- 面的压缩：由视线与物体边缘相切程度(Tangency)产生；

三、技术实现

1、颜色相接





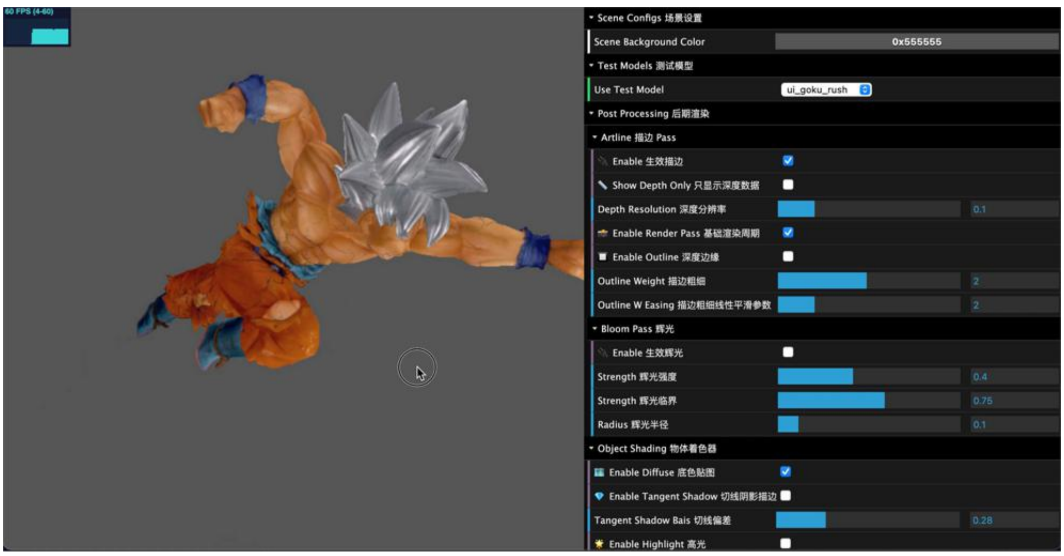
颜色相接产生的线条，是 3D 渲染中的固有属性，固有属性的线条不会因为光线或视角改变突然出现或消失；

2、深度断层

3D 场景中，在渲染所及之处都会有一个对应的坐标，以深度图像方式呈现。下图是从 3D 到 2D 的降维转换，距离相机远的物体信号比较弱，以黑色表达；距离相机近的部分则以高亮表达。



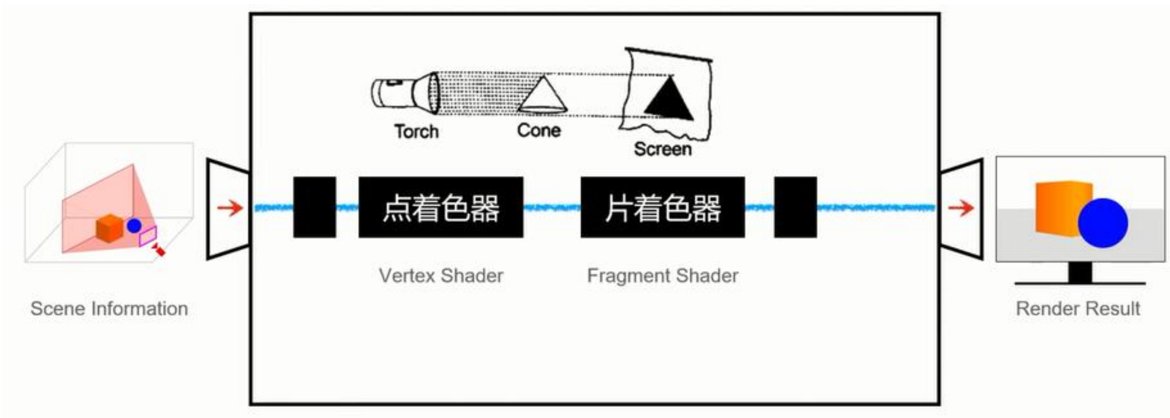
将 3D 模型加载到渲染器中，打开“深度”可以清楚看到距离的远近差别。



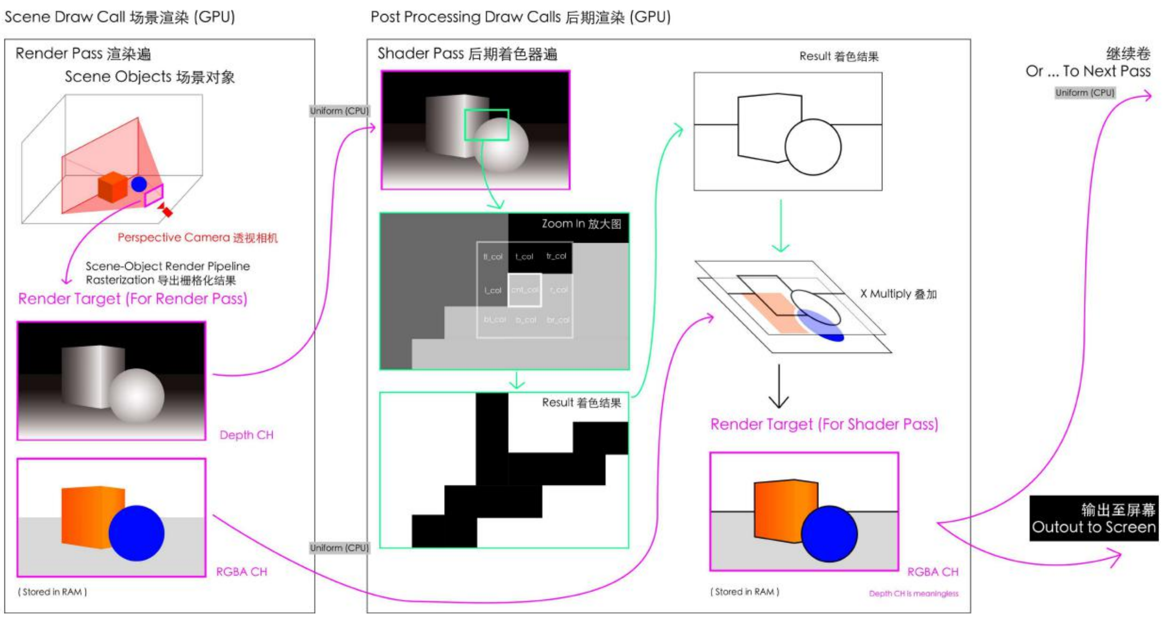
深度图像

单通道渲染管线

渲染通道的需求是将 3D 场景数据拍屏，输出成一张扁平的图片，类似用手电筒将 3D 物体投影到屏幕上。



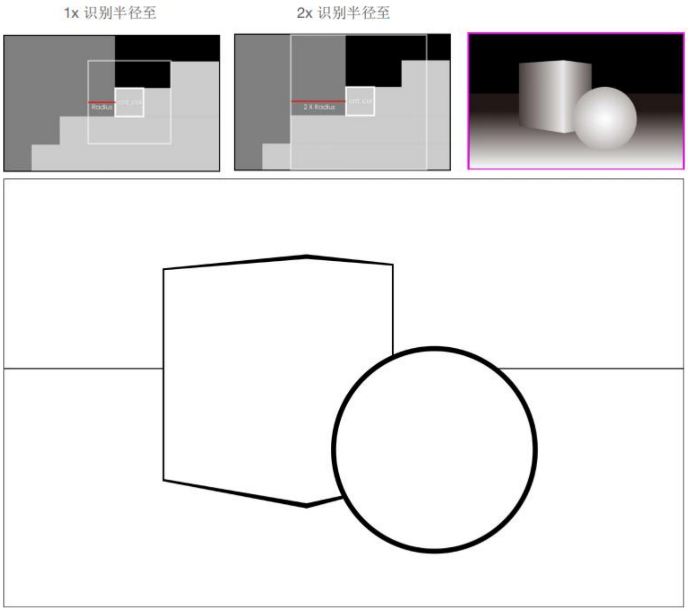
上图右边的渲染结果没有勾线，需要在执行前进行勾线操作，操作步骤如下图：



从场景出发，在第一次渲染后会产生 2 个结果：深度图像（Depth CH）和渲染结果（RGBA CH），将深度图像局部放大进入像素级别，与邻近像素做颜色对比，也就是深度对比，当深度达到预设定值就变成黑色，未达到变成白色，继而得到着色结果图，将相机拉远会得到一个粗略的勾线图，将勾线图与颜色图像进行叠加就可以得到一张勾线图。

关联深度与线的粗细

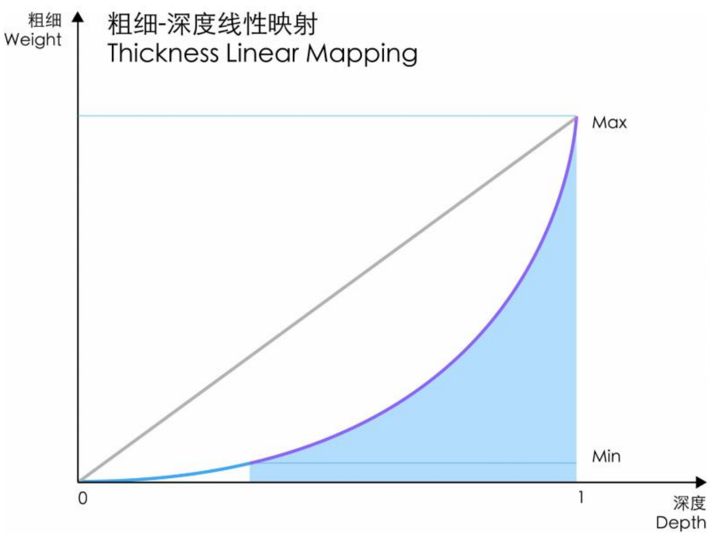
如何将线渲染得有灵魂，可以从线的粗细入手。线的粗细由邻近像素对比的半径决定，将其与深度图像进行关联后，可以得到优化图（如下图），在优化图中可以看出物体的深度。

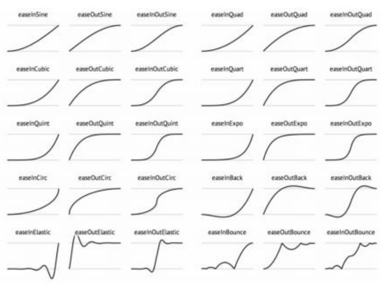


优化图

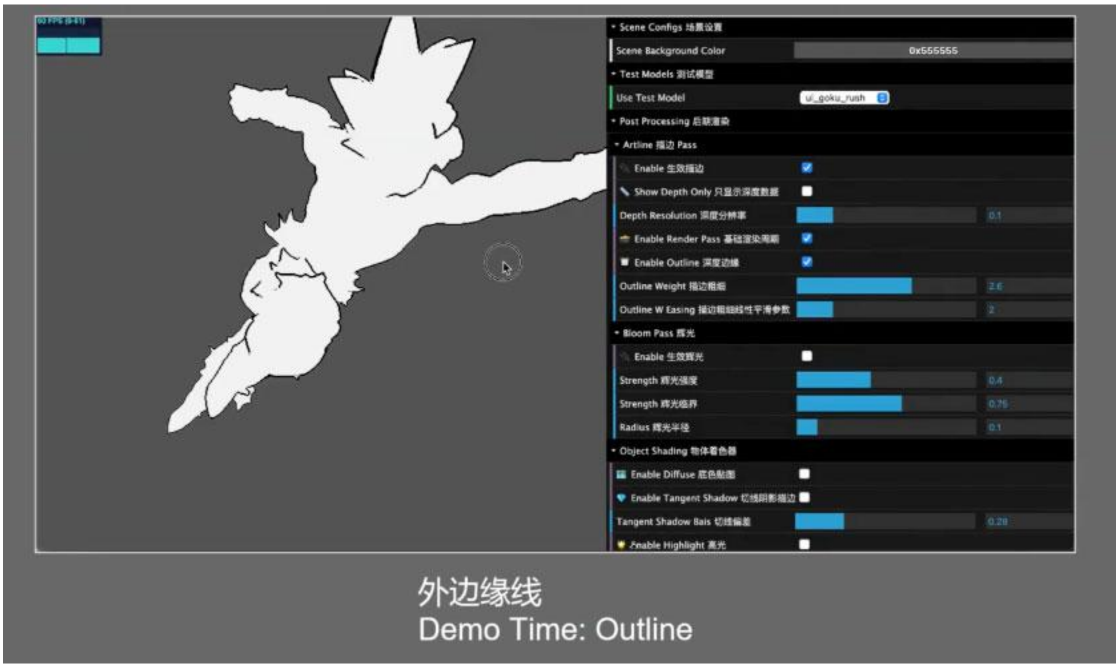
粗细-深度线性映射

将直线性的深度变化映射到曲线上，这样既可以增加线条的灵动感，又可以控制线粗细的最小值，在渲染时不至于丢失距离远的物体。



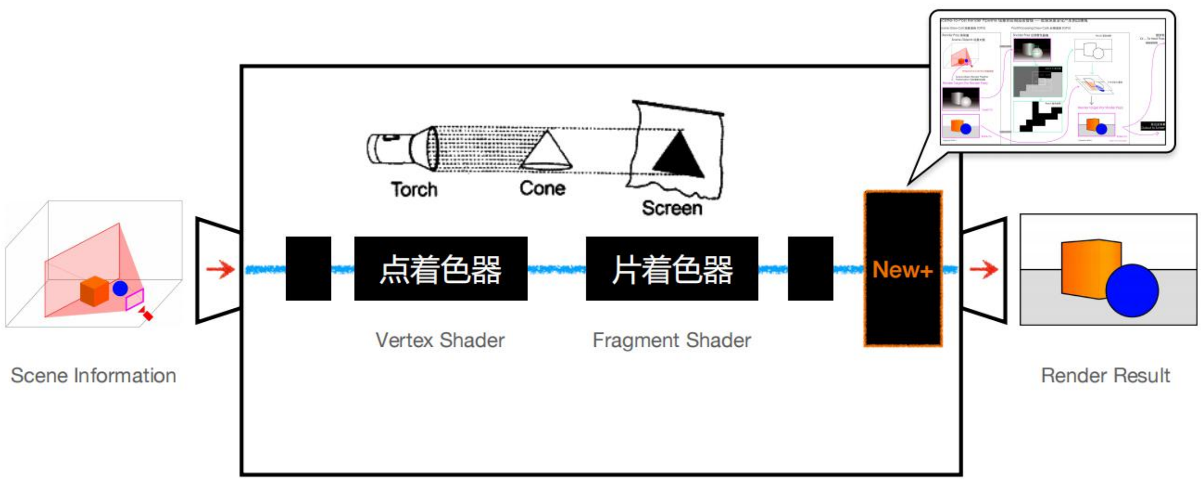


Demo：勾线效果

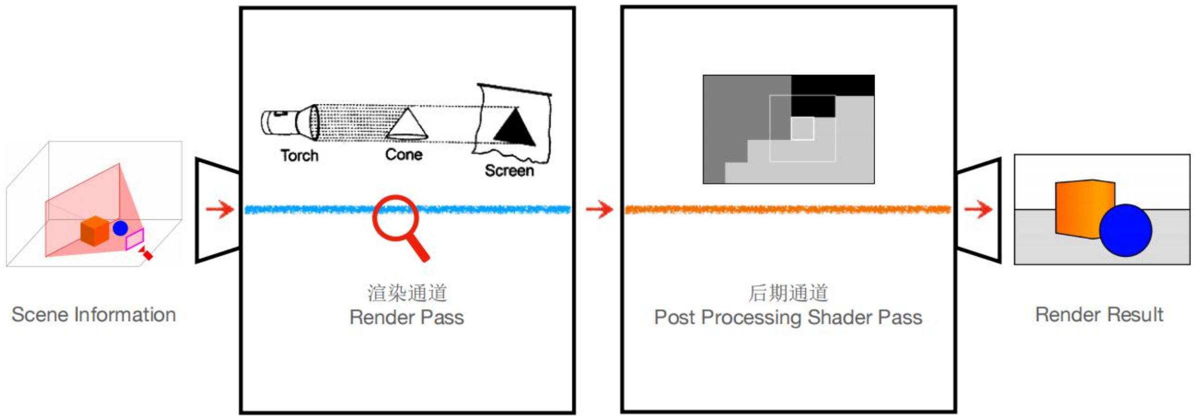


多通道渲染管线

将修改后的渲染管线进行整理后，得到双通道渲染管线，第一个通道是原生渲染通道，第二个后期通道则是后期勾线。



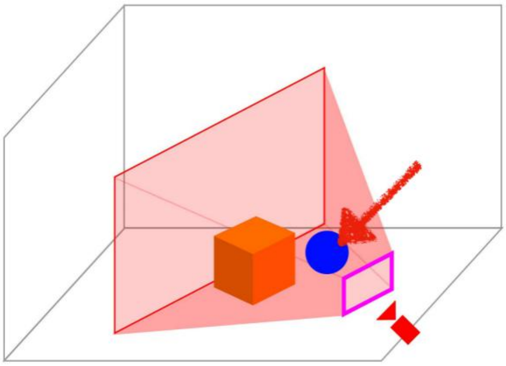
修改后的渲染管线



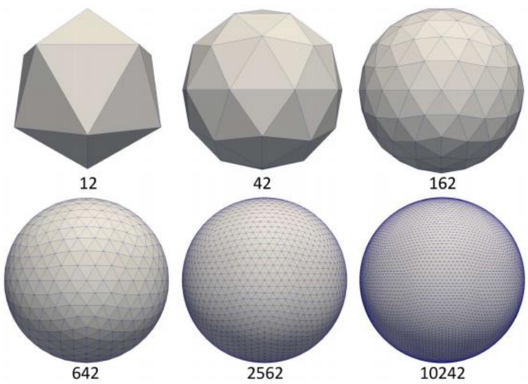
双通道渲染管线

3、面的压缩

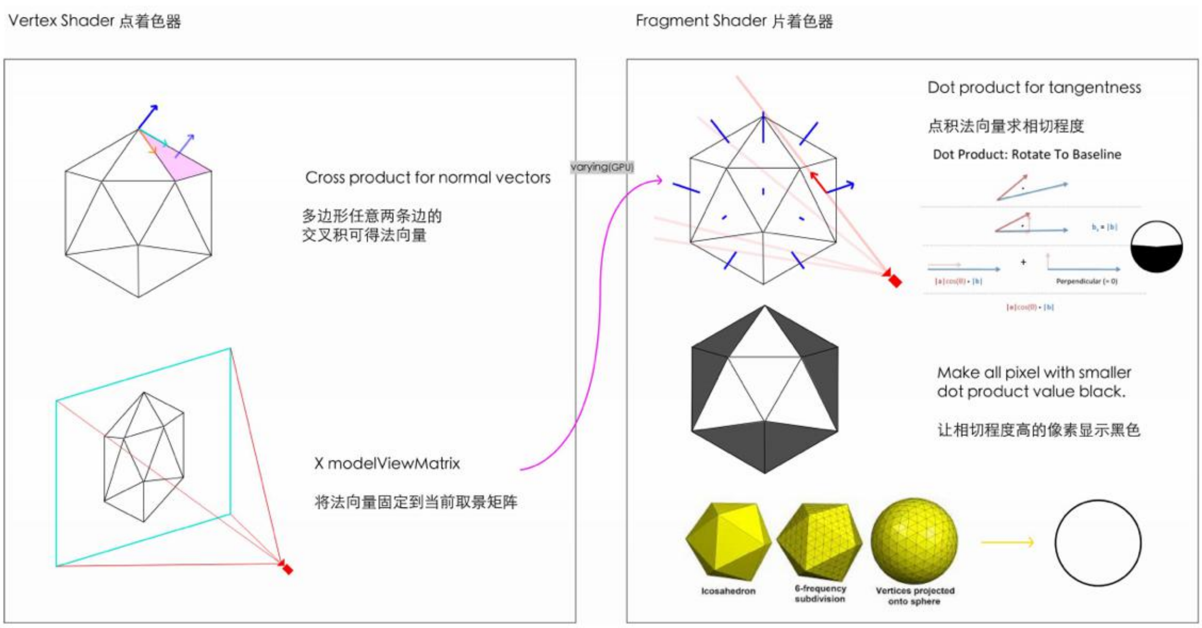
面的压缩由视线与物体相切程度(Tangency)产生。



将场景中的球单独拿出进行简化，在保持球的对称性基础上将其简化成 12 面体，这样可以清楚的看到每个面的朝向。

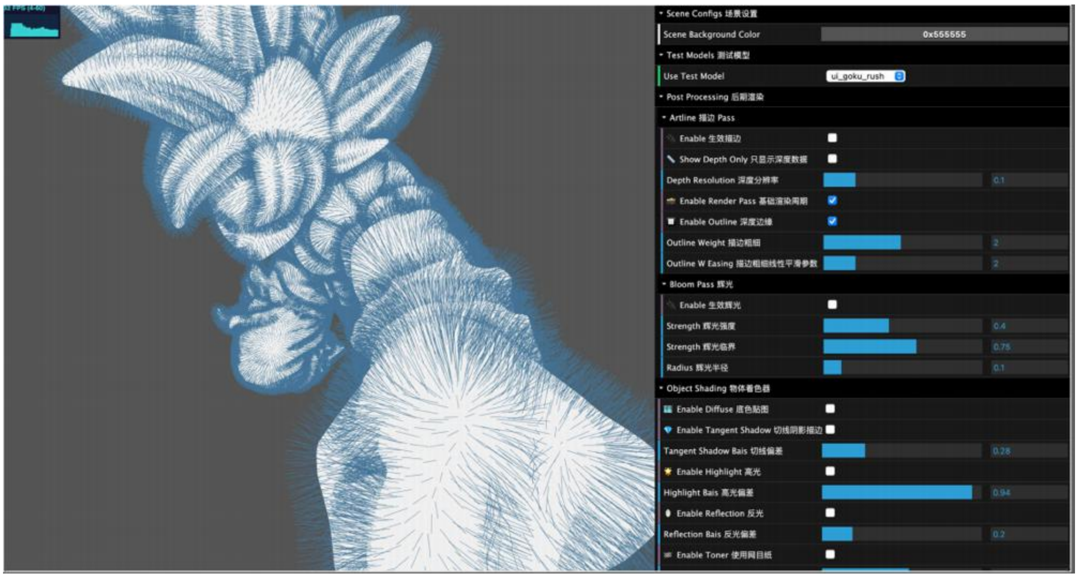


3D 图像面的朝向是由其法向量来表示，由于法向量在物体旋转或视角切换时会失效，因此需要先将法向量固定到当前取景矩阵，然后将视线与面朝向的法向量对比，使用点积法向量求相切程度，让相切程度高的像素显示为黑色，当提高精度后会呈现出完美的勾线。



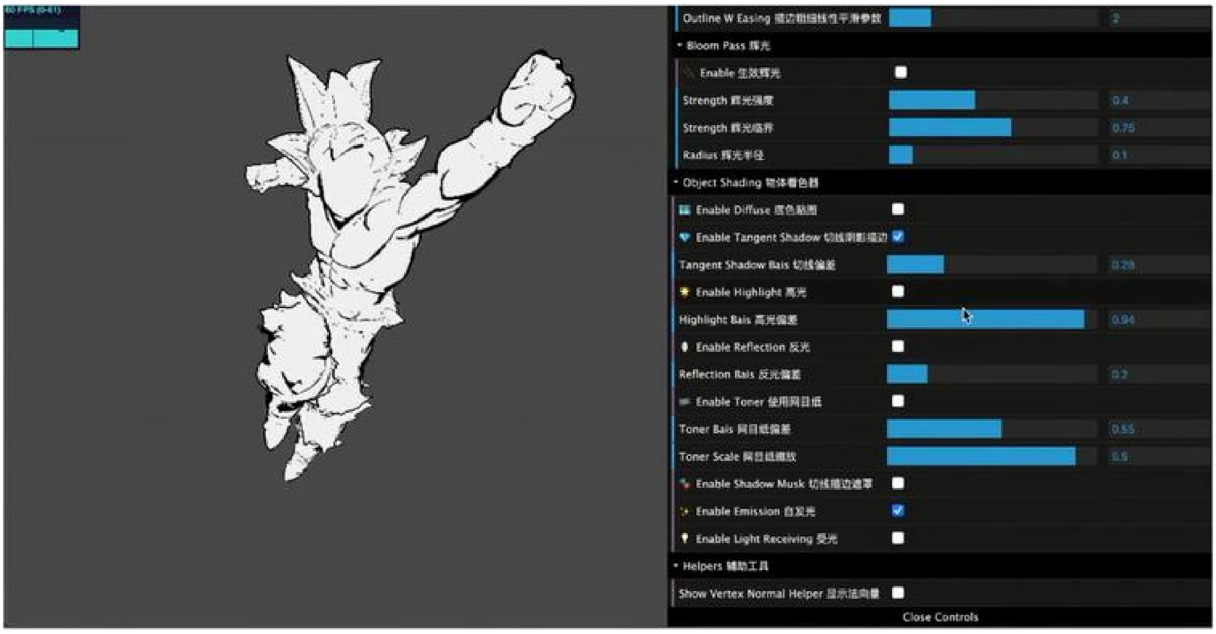
场景对象渲染管线-处理切线产生的线条

Demo：法向量可视化，其中每条线段表示一个面的朝向，将我们的视线与这些向量进行对比，来决定哪里需要画线。



法向量可视化

Demo：细节线，运用法向量相切的方式使线条更加丰富。



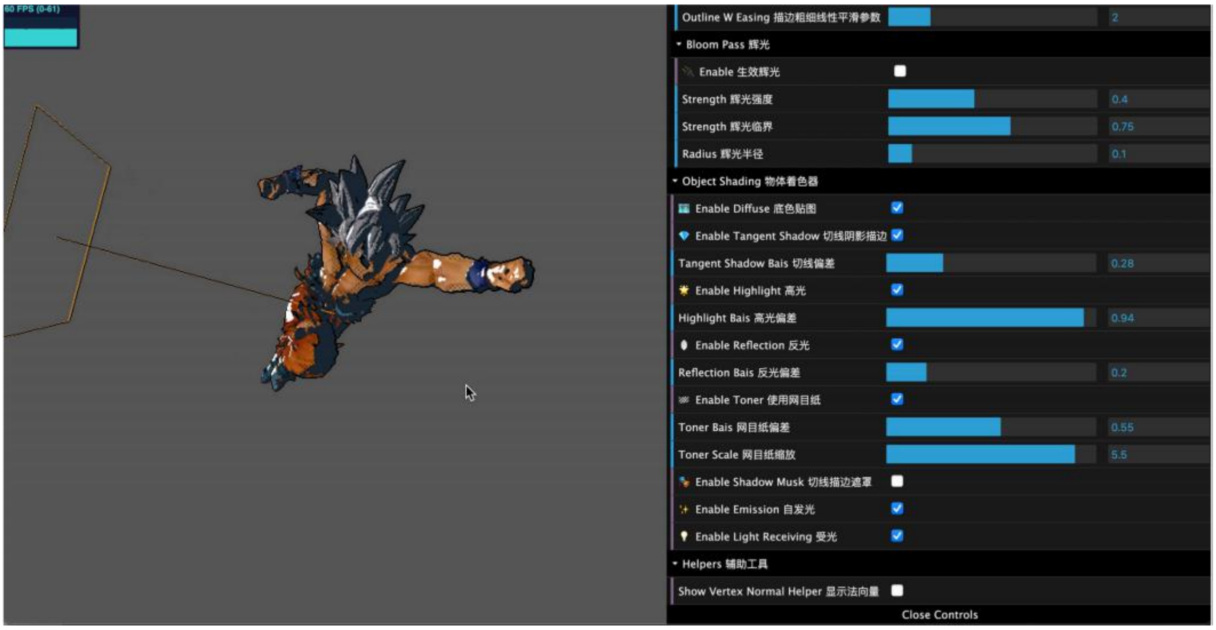
细节线

4、光源主导的分区着色

当光源与视线相切，可以根据不同点积值来区分 Cel 分区。

比如：

- v <= 0.01 边缘线
- v <= 0.1 暗部
- v <= 0.15 次暗部
- v <= 0.25 固有色
- v >= 0.95 高光



Demo：光源主导的分区着色

Demo 完整版：Shader 沙箱 APP 开源地址

https://github.com/Rubinhuang9239/shader-test-manga

开源表单方案 Formily 的核心设计思路

摘要： Formily 是一个数据+协议驱动的表单解决方案，它站在 Reactive 响应式编程巨人的肩膀上，构建出了从基础表单到低代码领域的高性能通用基础能力，同时其配套的跨框架+跨终端组件生态体系，也能让用户更高效的开发日常业务表单，尽可能的减少了重复冗余的逻辑实现。本话题将与你分享如何在高复杂业务场景下提高我们的表单性能与表单开发效率。

分享人： 王智力(白玄)，阿里巴巴 Formily 开源项目核心架构师。

一、什么是 Formily?

Formily 是一款面向中后台复杂场景的数据+协议驱动的表单框架，它也是阿里巴巴集团的统一表单解决方案。借助 Formily 可以完成复杂的表单页面需求，同时 Formily 提供的表单设计器可以快速搭建表单。

Formily 官网：https://formilyjs.org

1、核心特点：

- 高性能：Formily 在字段数量无限多和高频输入场景下，可以保证 O（1）时间复杂度；
- 框架无关：Formily 已经完美兼容 React、Reactive、Vue 框架；
- 生态完备：Formily 在组件桥接层支持 Ant Design、Element、阿里巴巴 Fusion 组件库；
- 协议驱动：协助 Formily 让后端通过 JSON 动态驱动表单渲染以实现表单可搭建可配置；
- 相当完备：Formily 可以支持各种复杂的表单页面需求。

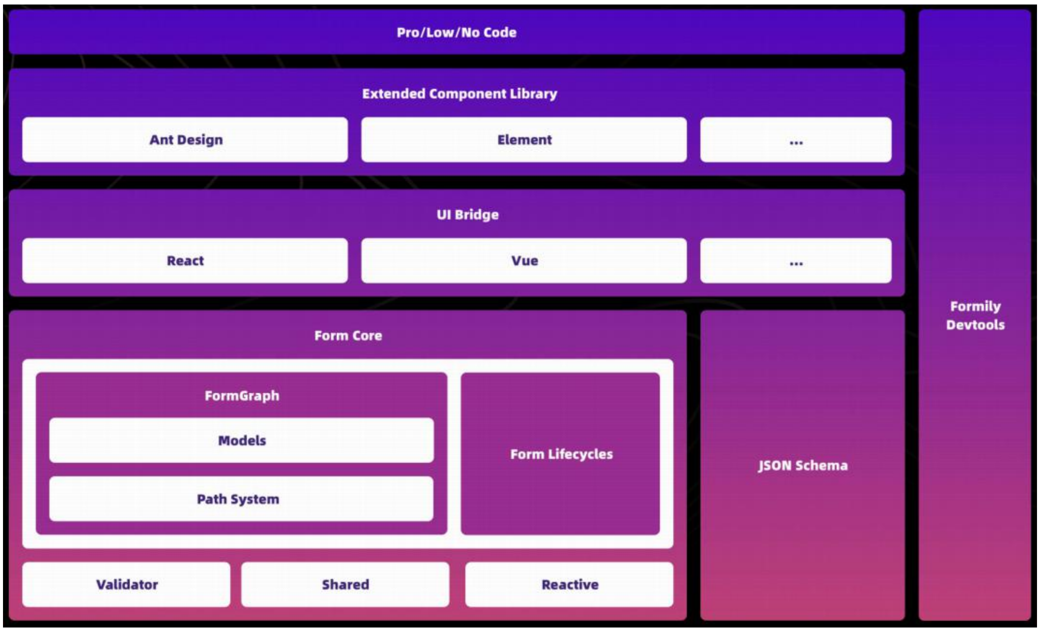
2、设计原则

- 单一职责；
- 优雅命名：遵循简单、直观、一致的标准。

3、核心架构

Formily 核心架构分为四层，自下而上分别是：

- 内核和协议层：内核是表单相关 UI 无关的领域模型系统；协议层是基于 JSON Schema 扩展的表单协议；
- UI 桥接层：基于 Formily 内核与各种 UI 框架（如 React、Vue）桥接胶水层；
- 组件扩展层：建立在 UI 桥接层与 UI 组件库的桥接胶水层；
- 应用层：借助 Formily 可以构建纯源码、低代码、无代码等应用解决方案；
- Formily 开发工具（下图右侧）



Formily 核心架构

二、设计思路

1、高性能思路

问题：普通 React 用法，想要实现数据收集和同步，需要强依赖虚拟 DOM 的整树重绘才能达到目的。

目标：不管字段数量多少，高频输入永远都是 O(1)复杂度。

方案一、ReactFinalForm/Antd4.0 方案

抽象 FormState/FieldState，内部使用 pub/sub 模式进行通讯，脏检查控制联动时的渲染重绘。

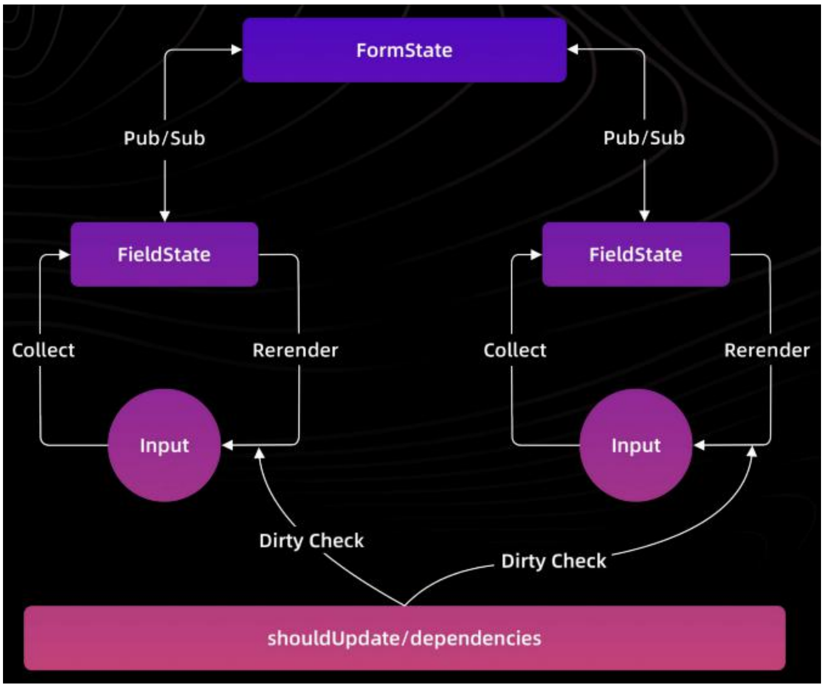
时间复杂度：

输入时：Ant Design O(1)/ReactFinalForm O(1)

联动时：Ant Design O(1 or n)/ReactFinalForm O(1 or n)

优点： 简单，易上手；

问题：脏检查成本高，外部联动无法做到 O(1)；



方案二、UForm/Formily1.x 方案

抽象 FormState/FieldState，内部使用 pub/sub 模式通讯，抽象主动更新模型，可以解决方案一中外部联动无法做到 O（1）的问题。

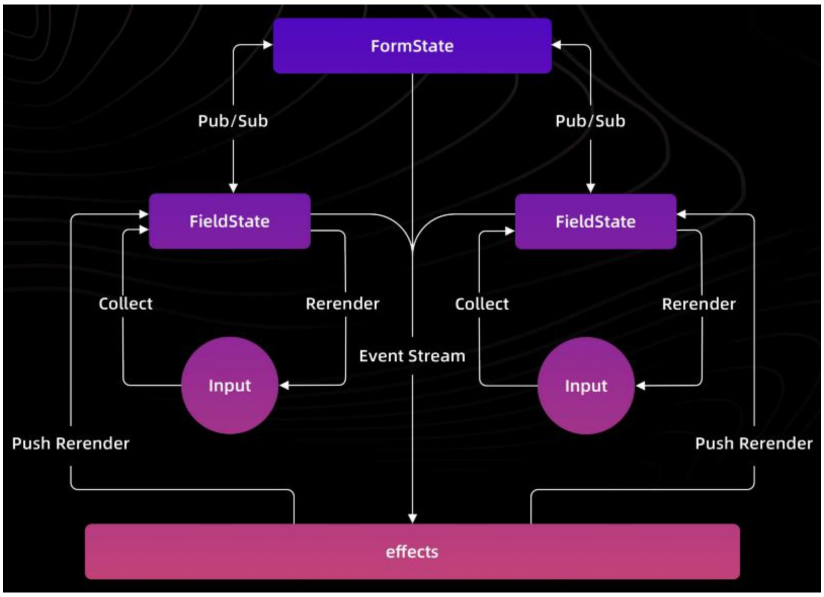
时间复杂度：

输入时：O(1)

联动时：O(1)

优点：高性能；

问题：学习成本高，外部联动需要转换成主动模型；



方案三、ReactHookForm 方案

基于 DOM 机制收集数据，绕过 React 受控更新机制，保证高频输入无卡顿。

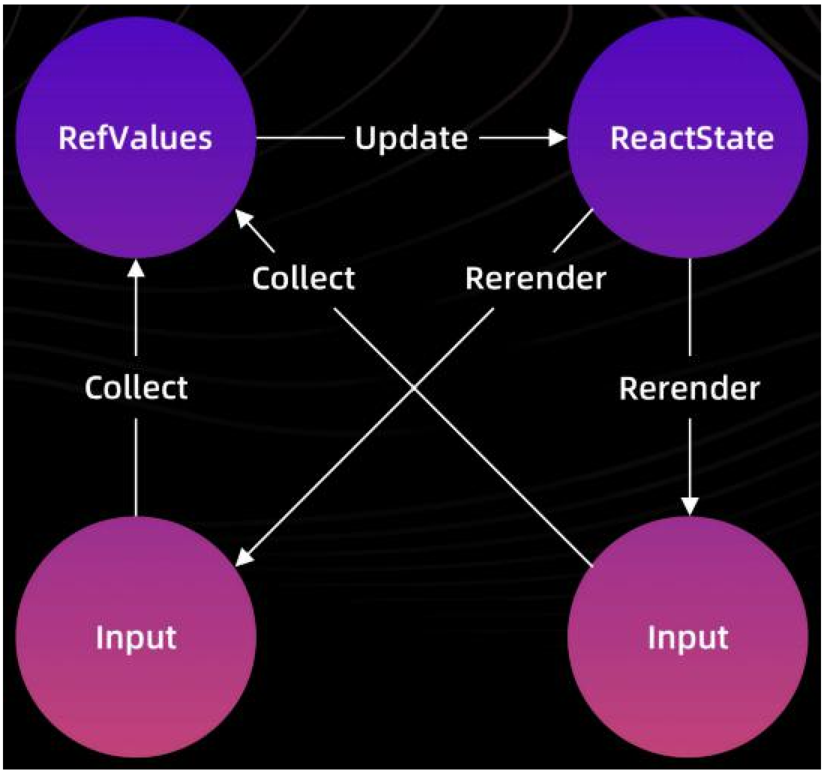
时间复杂度：

输入时：O(1)

联动时：O(n)

优点：足够轻量，简单；

问题：一旦涉及数据联动，还是需要重绘整树；



方案四、Formily2.0 方案

依托 Reactive 响应式解决方案，构建整个响应式表单领域模型。当组件所依赖的响应式变量发生任何变化，组件都会重新渲染，从而解决组件的脏检查问题。

时间复杂度：

输入时：O(1)

联动时：O(1)

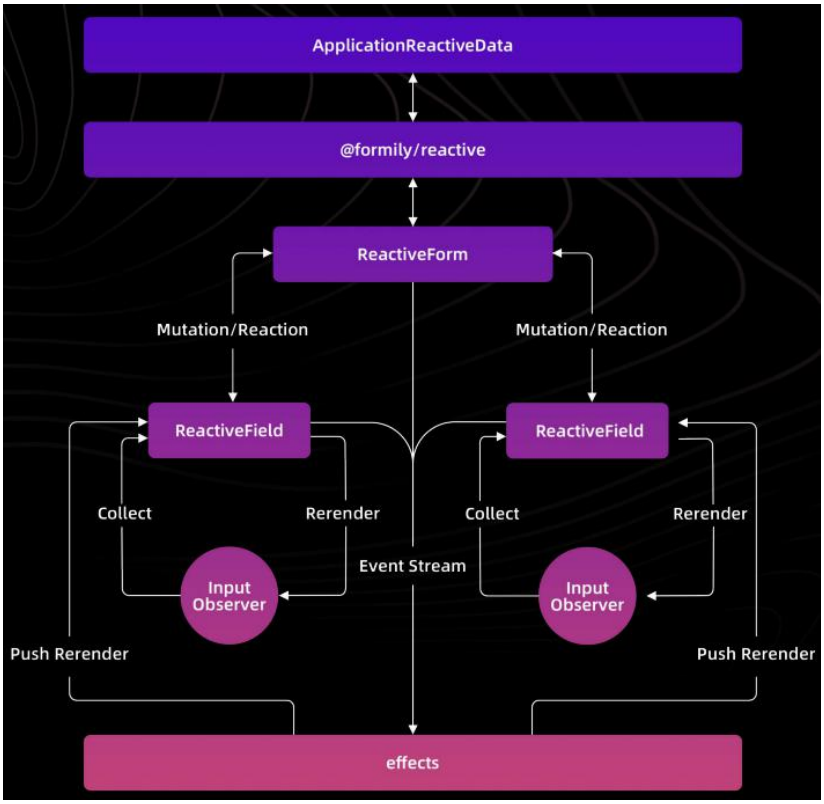
优点：

不再需要大量脏检查成本，一切都是基于数据变化精确渲染；

既支持主动更新模型，也支持被动响应模型，在多对一被动联动场景，代码量大大减少；

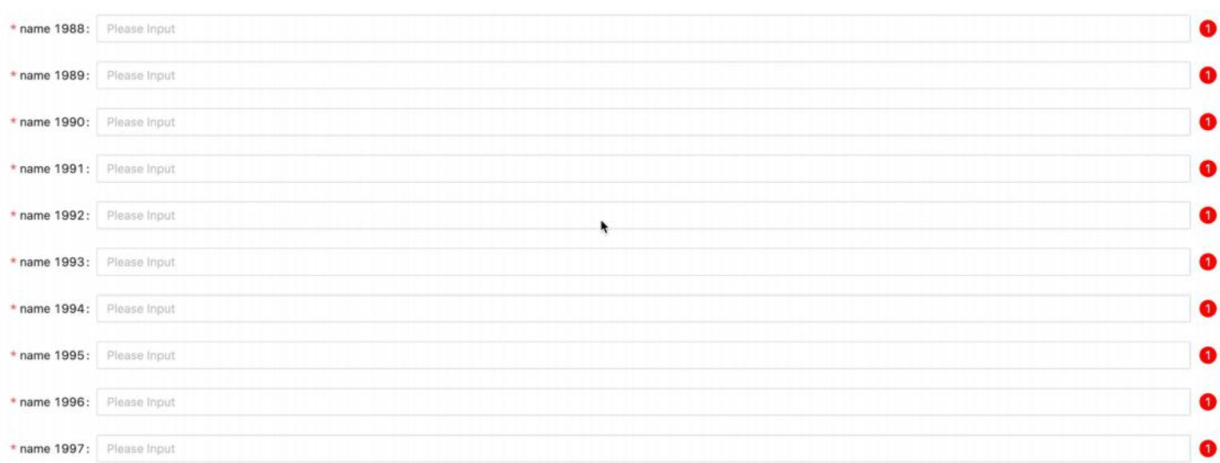
外部数据与表单内部字段联动时，只需要定义 ApplicationReactiveData 即可，代码编写成本很低；

问题：学习成本高。

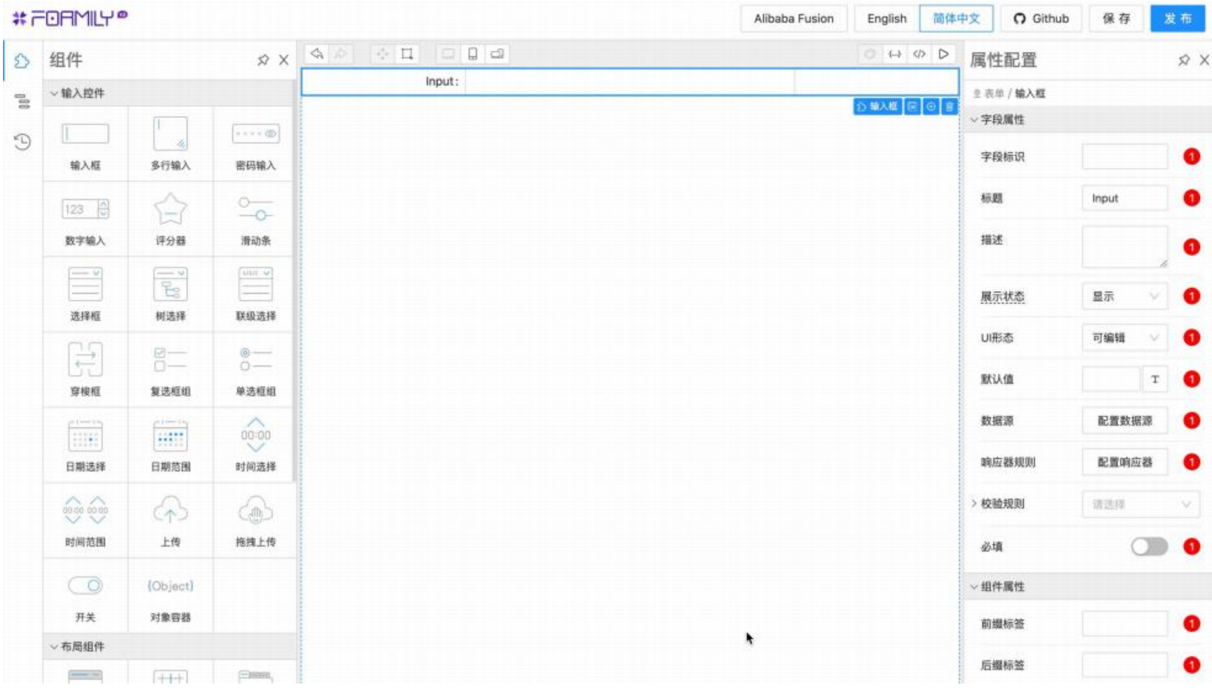


演示：高频输入性能和外部联动

时间复杂度 O(1) 复杂度，2000 个字段，针对单个字段的高频输入，完全无卡顿。



高频输入性能演示



外部联动演示

2、协议驱动思路

问题：如何更高效的开发表单？

目标：定义一套通用协议，简单高效的描述表单 UI/逻辑，并对低代码搭建友好。

方案一 纯 UITree 协议方案，类似于 JSON 版的 JSX

优点：

足够简单，componentName/props/children 完全可以把整个 UI 结构给描述清楚；

对搭建非常友好；

问题：

只能描述视图结构，无法描述状态模型。

```
{
  componentName:"Form",
  props:{},
  children:[
    {
      componentName:"Field",
      props:{
        name:"username"
      },
      children:[
        {
          componentName:"Input",
          props:{
            placeholder:"Please Input"
          }
        }
      ]
    }
  ]
}
```

方案二 标准 JSON Schema 方案，参考 ReactJSONSchemaForm

优点：

以数据描述视角来驱动 UI 渲染，基于这样的数据描述，可以将很多状态处理逻辑固化到渲染引擎或者组件内部，从而使得协议层变得非常干净；

问题：

无法解决 UI 容器包裹问题，对搭建友好。

```
{
  type:"object",
  properties:{
    array:{
      "title": "Custom array of strings",
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
```

方案三 Formily2.0 方案，针对 JSONSchema 的扩展协议

核心思路：

扩展 Void 类型描述数据无关布局容器或控件；

扩展 x-* 属性描述 UI 控件与控件属性；

优点：

完全表达 UI，无需第二套协议，学习成本很低；

可以将临时状态封装在渲染引擎或者组件内部；

对搭建更加友好，因为它已经可以把 UI 层级给表达出来了；

```
{
  "type":"object",
  "properties":{
    "card":{
      "type":"void",
      "x-component":"Card",
      "x-index":0,
      "properties":{
         "categoryId": {
            "type":"string",
            "x-component":"Input",
            "x-index": 0,
         },
         "brandId":{
           "type":"string",
           "x-component":"Text",
           "x-component-props":{
              "style":{
                "color":"red"
              }
           },
           "x-visible":"{{$values.categoryId === 2}}"
         },
      }
    }
  }
}
```

3、扩展组件思路

问题：如何开发协议思路的自定义组件？

目标：足够简单，符合标准组件开发最佳实践。

基础扩展方案

属性约定，比如 value/onChange 组件属性约定，保证所有输入型组件可以快速接入。

优点：

符合统一 React 属性命名规范标准，对于第三方组件接入成本很低；

缺点：

只能约定标准属性，value/onChange/disabled/readOnly 之类的，无法约定渲染引擎内部与扩展组件间其他数据交互。

```
const CustomComponent = ({value,onChange})=>{
  //渲染引擎会给自定义组件传字段值 value
  //自定义组件内部数据发生变动，可以通过 onChange 告诉渲染引擎字段值发生变化
}
```

白盒进阶扩展方案

标准 Hooks 方案，借助 ReactContext，就近寻找渲染引擎内部状态模型。

优点：

完全隔离了渲染引擎与组件自身属性，做到了类型更友好，写法更自然的通信模式；

缺点：

对接三方组件每次都需要单独写一个包装组件，不方便快速桥接。

```
const CustomComponent = ({value,onChange})=>{
  const field = useField() //就近寻找当前字段模型
  const schema = useFieldSchema() //就近寻找当前字段Schema描述
}
```

黑盒进阶扩展方案

连接器扩展，与白盒进阶扩展不同的是黑盒用 connect 和 mapProps 做了一层包装。

优点：

无需重新封装组件，只需要关注属性与字段模型的映射；

映射转换逻辑方便复用；

缺点：

学习理解成本高。

```
import { connect, mapProps } from '@formily/react'
import { Select as AntdSelect } from 'antd'

export const Select = connect(
  AntdInput,
  mapProps(
    {
      //简单属性映射
      dataSource: 'options',
      loading: true,
    },
    (props, field) => {
      return {
        //复杂属性映射
        ...props
      }
  })
)
```

高级场景化扩展方案

渲染权限转移模板化方案。Formily 提供 RecursionField 组件负责动态渲染子树。

优点：

可以封装出大量模板组件，模板组件内部可以将布局、交互、逻辑全部固化，在实际使用过程中提效显著，比如 ArrayTable/ArrayCard 这种自增组件；

缺点：

学习理解成本高。

```
//过去：渲染引擎负责渲染子树
const CustomComponent = ({children})⇒{
  return (
    <Other>
      {children}
    </Other>
  )
}

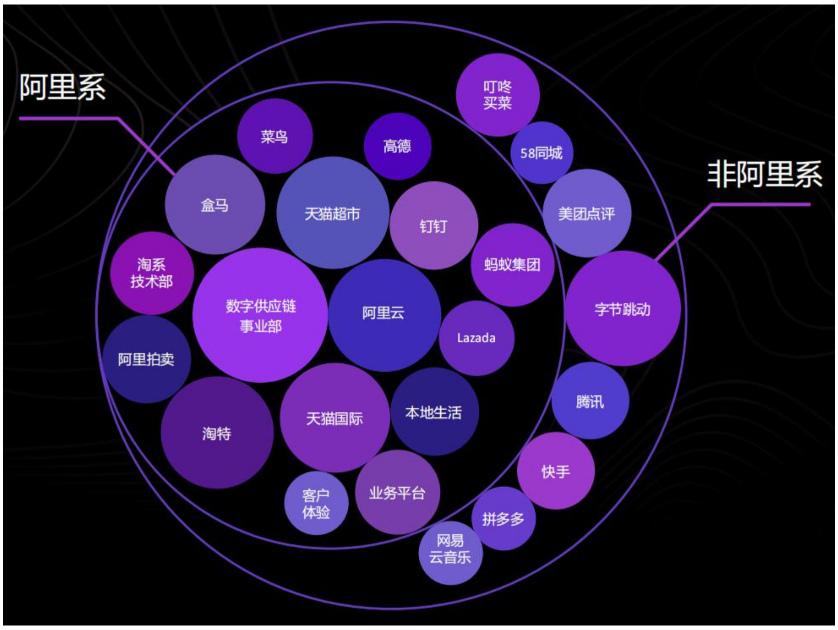
//现在：组件负责渲染子树
const CustomComponent = ()⇒{
  const schema = useFieldSchema() //就近寻找当前字段Schema描述
  const { positionA, positionB, positionC } = parseSchema(schema) //人工解析schema
  //插槽化渲染，布局，交互，逻辑固化
  return (
    <Other>
      <RecursionField name="positionA" schema={positionA} />
      <RecursionField name="positionB" schema={positionB} />
      <RecursionField name="positionC" schema={positionC} />
    </Other>
  )
}
```

综上所述，Formily 的扩展能力是建立在标准组件开发最佳实践基础之上，是没有任何黑魔法的。

三、使用数据

1、使用数据情况

如下图所示，Formily 被广泛应用于阿里系和非阿里系的各种 BU 的不同场景。



2、社区运营数据



四、未来规划

1、规划

- 进一步优化性能；
- 构建更多的扩展生态；
- 支持 React18；
- 思考如何借助 Formily 一次性解决 CRUD 页面；

2、不足

- 不兼容 IE，且无法解决；
- 整体体积较大，压缩前 130k 左右，压缩后 80k 左右，包含桥接组件库；
- 学习成本比较高，计划逐步完善视频教程。

小结：Formily 的价值

Formily 在高性能的加持下，实现一套底层适配多个框架，一份 Schema 适配多端（PC、手机），一个生态覆盖全场景（Pro/Low/No Code）。

更多 Formily 相关信息和分享请关注钉钉 Formily 技术群，有问题或者希望进一步交流的朋友欢迎联系。



协同文档工作机制简介

摘要：随着在线办公的兴起，传统办公套件 Office 的在线化需求也随之增加。钉钉文档作为钉钉核心办公套件之一，上线已经三年，其间持续迭代，已成为一个极其复杂的产品。对前端工程师而言，协同文档是一个较为有挑战的领域，除了传统天坑富文本编辑器外，还引入了协同编辑这一挑战，钉钉文档甚至还支持专业排版能力。本话题将会以钉钉文档为例，简述协同文档的工作机制。

分享人：崔晓斌(本杰)，钉钉文档前端架构师，前 Ant Design 核心维护者，GitHub benjycui

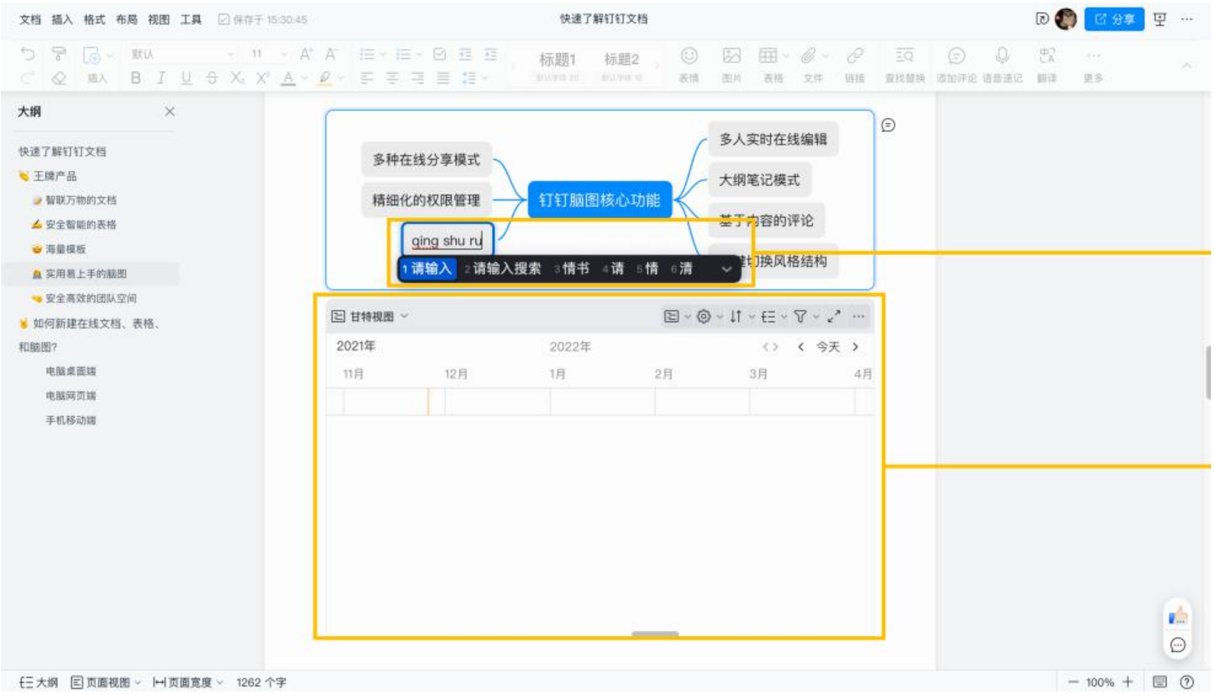
一、关于钉钉文档

钉钉文档是阿里巴巴钉钉推出的、与钉钉深度结合的在线文档产品。如下图所示，钉钉文档具有富文本编辑和协同编辑能力。



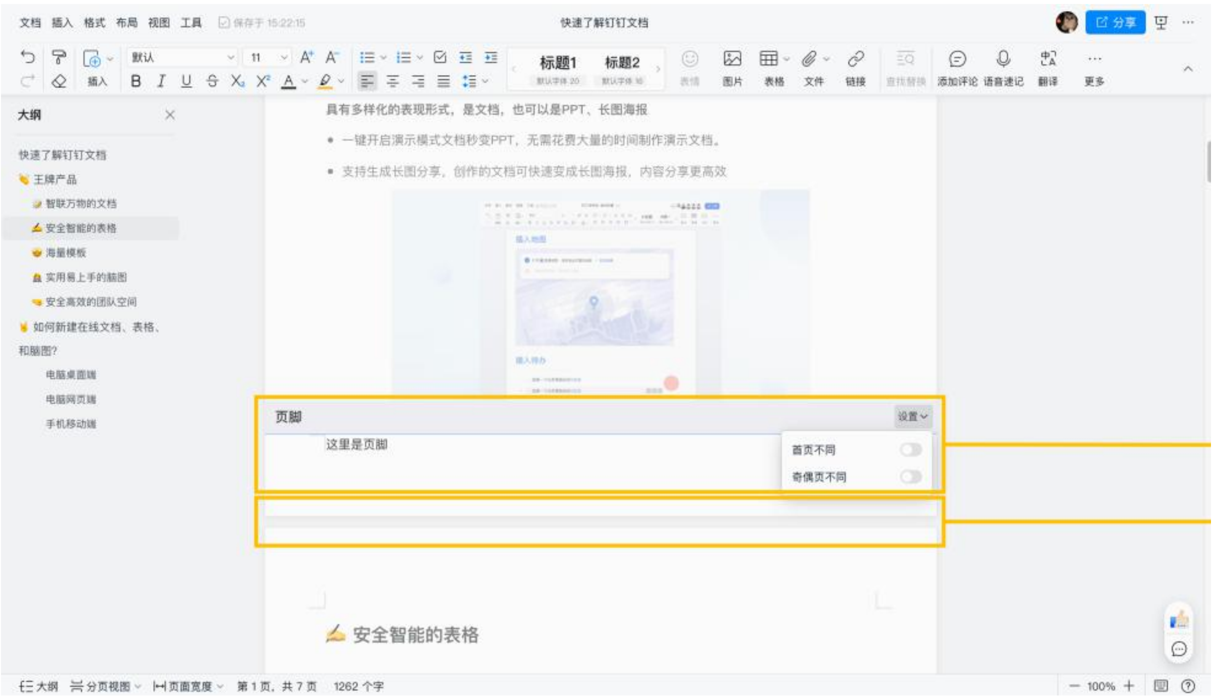
（黄色线框从上至下依次为：富文本编辑、批注、协同编辑）

钉钉文档支持嵌入脑图、甘特图等区块，且嵌入区块可协同编辑。



（区块可协同编辑，超过 20 种区块）

与其他富文本编辑器比较，钉钉文档的强大之处还体现在：支持专业排版功能，包括：分页、分栏、图文混排等，因此在公文、打印的场景有更加友好的表现。



（页头、页脚，分页）

钉钉文档的核心性能有：

- 专业排版能力；
- 不依赖 contentEditable 编辑器；
- 协同编辑能力。

二、专业排版

同一个文档在钉钉文档中既可以以流式视图形式呈现，也可以以分页视图呈现，所见即所得，如下图：



流式视图　　　　分页视图

浏览器提供的排版能力只能支持流式视图，若要实现分页视图就需要同时具备文本的折行和拆分能力，而钉钉拥有专业排版能力。

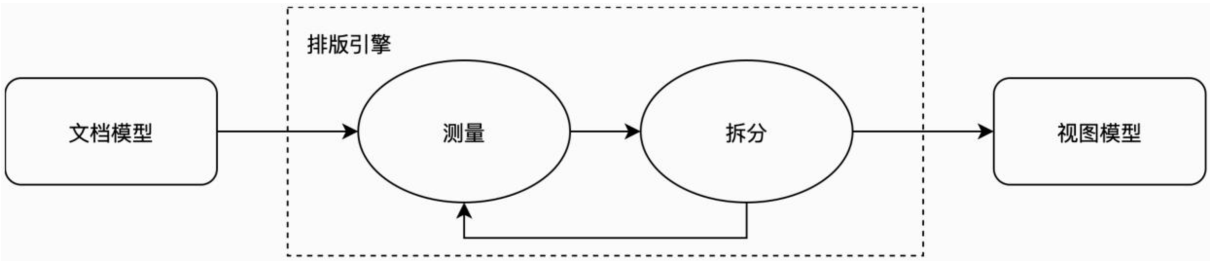
1、分行

如何分行是排版中最简单也是最基础的一个问题。一段文本在达到容器所能容纳的最大宽度后会触发换行。



2、测量与拆分

在钉钉文档中用户输入的内容会存储在文档模型中，而文档模型自身不包含任何布局信息，经过排版引擎处理后增加分行、分页等布局信息。排版引擎中最关键的两个部分是测量和拆分。



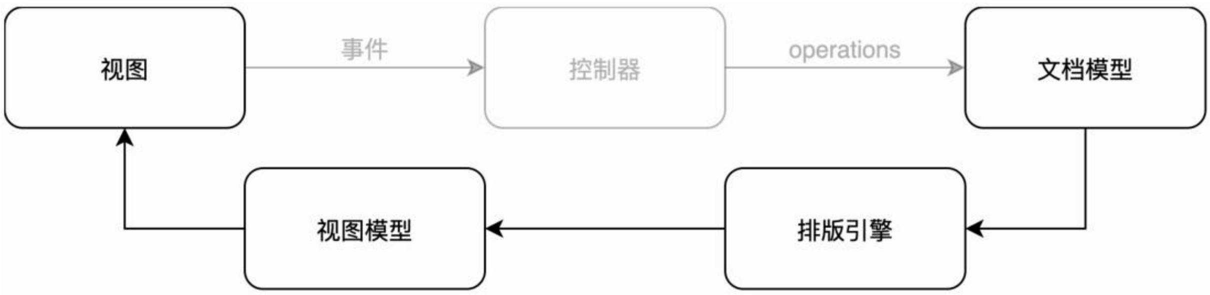
3、字符测量与分行点

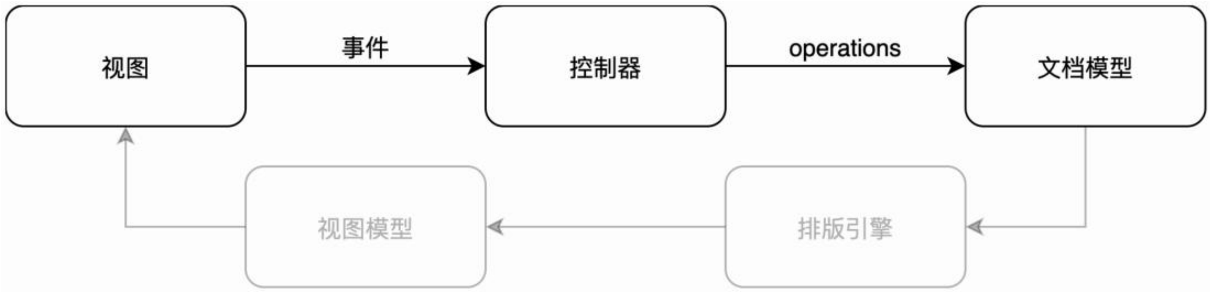
对字符逐一进行宽度测量并加总，再结合当前容器的宽度，可以得知分行点。



4、编辑数据流

用户在排版后的视图模型上交互，实际修改的是排版前的文档模型。





5、拆分与映射

通过约定数据节点唯一标识，及其映射关系，建立起视图模型到文档模型的链接。



如图所示，文档模型中的唯一标识是段落 1，视图模型中的唯一标识是根据特定规则从文档模型的唯一标识派生而来，段落 1-0 和段落 1-1 是段落 1 拆分出的两个段落。基于这个映射关系，可以推导出用户的编辑意图，如用户选中段落 1 的“行”并加粗，通过映射关系可以推出用户要编辑的是文档模型中段落 1 的第 22 个字符。

以上是钉钉文档如何支持专业排版的基本原理，以实现所见即所得的目的。

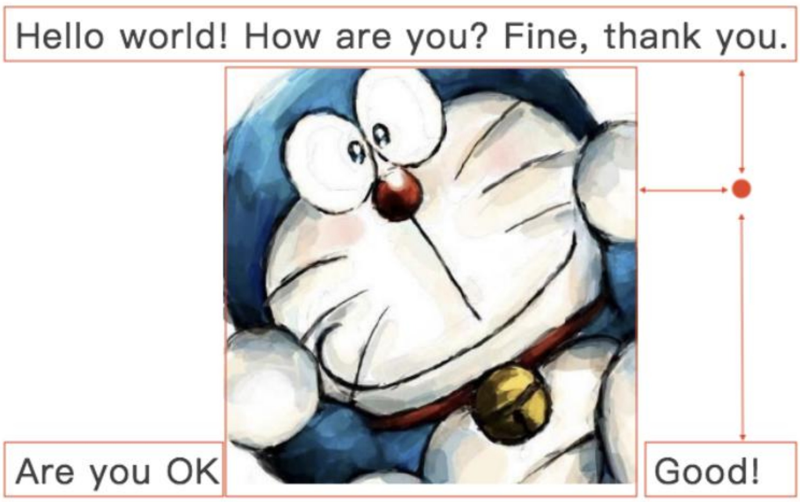
三、不依赖 contentEditable 的编辑器

contentEditable 是 W3C 提供的标准能力，它提供作用于 DOM 的编辑能力。然而钉钉文档需要专业排版能力，这就使 contentEditable 提供的视图模型与钉钉需要的文档模型并不匹配。因此，为了实现专业排版能力钉钉文档需要自己实现富文本编辑能力。这其中最关键的两个问题是：

- 不依赖 contentEditable，编辑器如何响应用户的点击行为并绘制光标？
- 不依赖 contentEditable，编辑器如何响应用户的输入行为并呈现字符？

1、视图 - 选区计算与绘制

以下图为例，当用户点击文档中空白部分，最符合用户直觉的行为是选中 Good 与!之间。



2、视图 - 事件响应

使用鼠标、触摸事件计算选择行为。

如下图代码所示，用户点击文档后，钉钉从浏览器获取点击的 target 和 XY 坐标，基于这些参数，可以计算出用户选择的哪个数据节点的第几个字符。

```
const { target, clientX, clientY } = event;
if ('target 自身是 void (如图片、视频) 节点') {
  '直接选中该节点'
} else if ('target 自身是文本节点') {
  '基于 clientX & clientY 二分查找所点击位置对应文本中第几个字符位置'
} else {
  '按特定策略平移 clientX & clientY, 找到 target 子树中最符合用户预期的 void 或文本节点'
  '以重新调整后的 target、clientX、clientY 递归计算'
}
```

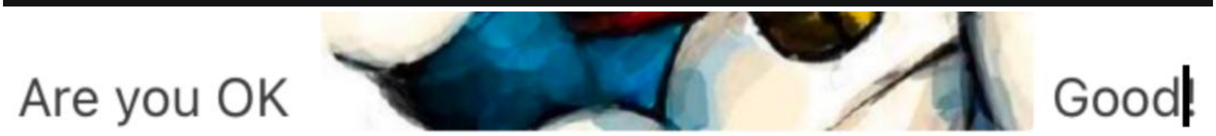


3、视图 - 选区数据描述

该数据结构除了被视图层消费以绘制光标外，也用于协同场景中协同光标的同步及绘制。

如下图代码所示，选区是 Value 中的 selection 字段，为了更友好的支持协同编辑，将选区进行数据描述并序列化成协同消息，广播给其他协同者绘制协同光标。

```
Value.create({
  selection: Selection.create({
    anchor: Point.create({ key: 'Good', offset: 4 }),
    focus: Point.create({ key: 'Good', offset: 4 }),
  });
});
```



以上操作可以解决第一个问题，即响应用户点击和绘制协同光标。

4、视图 - 输入上屏

当用户输入文本时，需要实时看到所输入的内容上屏，并能选中所需的字符



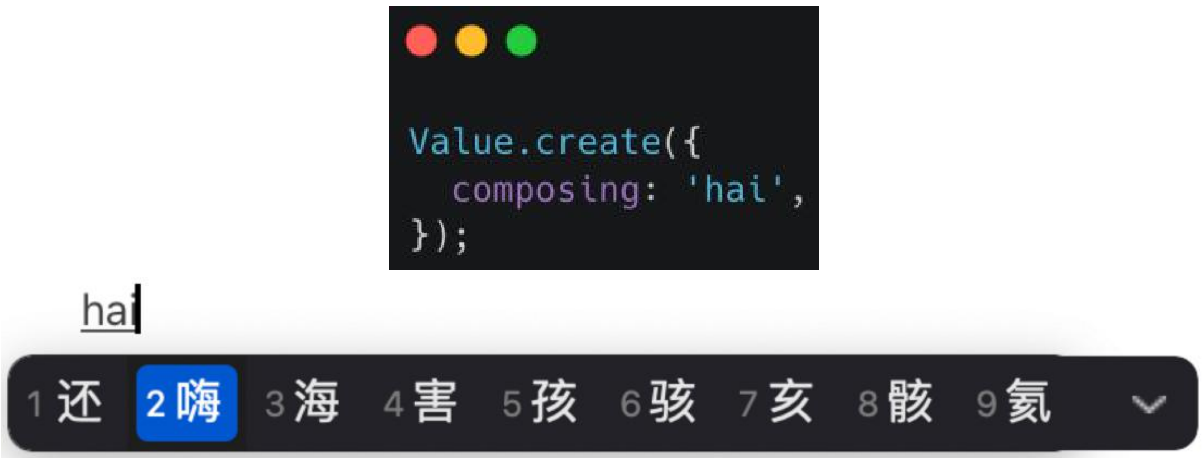
5、视图 - 幕后的 textarea

钉钉文档使用一个不可见 textarea 监听用户输入，并定位输入法浮层。



6、视图 - Composing 状态描述

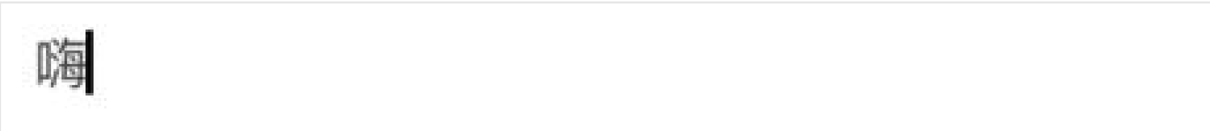
Composing 指中文输入开始到选词结束间的中间状态。



7、视图 - 内容数据描述

用户选词后，文本将会被插入到当前的文档模型。

```
Value.create({
  document: Document.create({
    nodes: [Paragraph.create({
      nodes: [Text.create('嗨')],
    })],
  });
});
```



8、视图 - 渲染为 DOM

用户选区、文档内容，经过排版后，使用 React 渲染并输出为 DOM。

```
<editor>
  <content {value.document + value.composing} />
  <selection {value.selection} />
</editor>
```

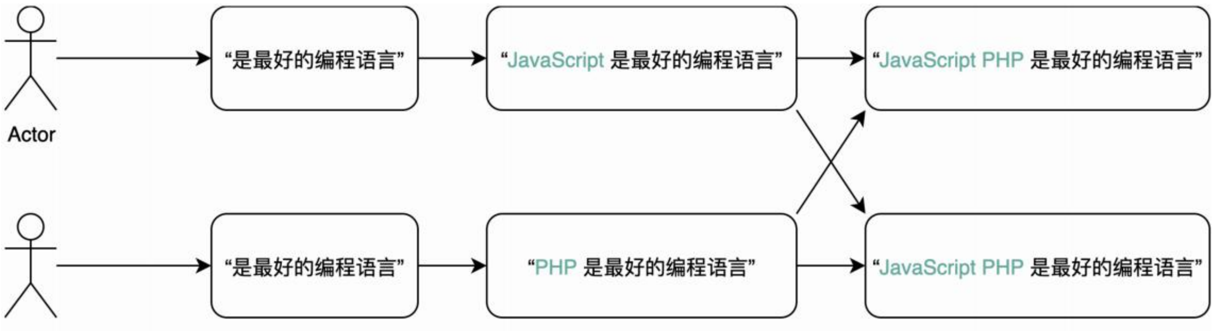
以上操作遵循将所有编辑器状态变成数据描述的基本设计思路，以便更友好的支持协同编辑。

四、协同编辑

钉钉文档的使用场景常常是将一个文档发到群里，由多个用户同时使用进行编辑，由此就形成了协同编辑。

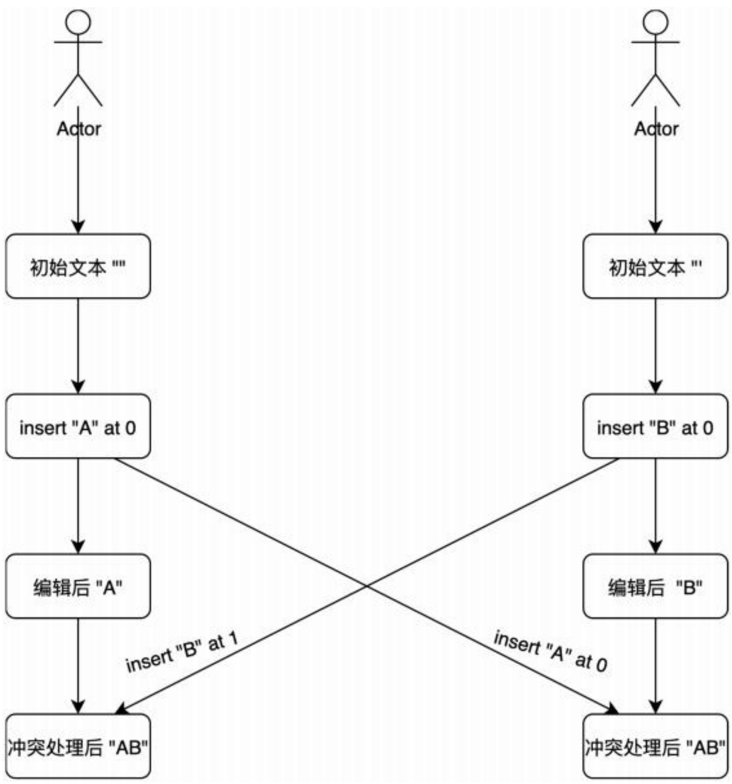
1、冲突处理

自动处理用户协同编辑所产生的冲突，保证最终数据一致。



2、Operational Transformation (OT)

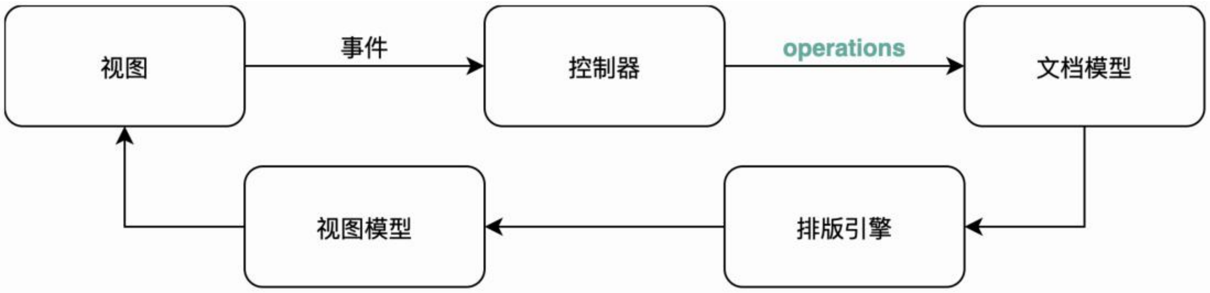
为实现上述自动处理冲突，钉钉后台使用 OT 算法理论。OT 算法理论的基本思路是将对文档模型的修改映射为差量数据 operation，在接收到他人 operation 时，使用 transform 处理潜在冲突。



3、编辑数据流

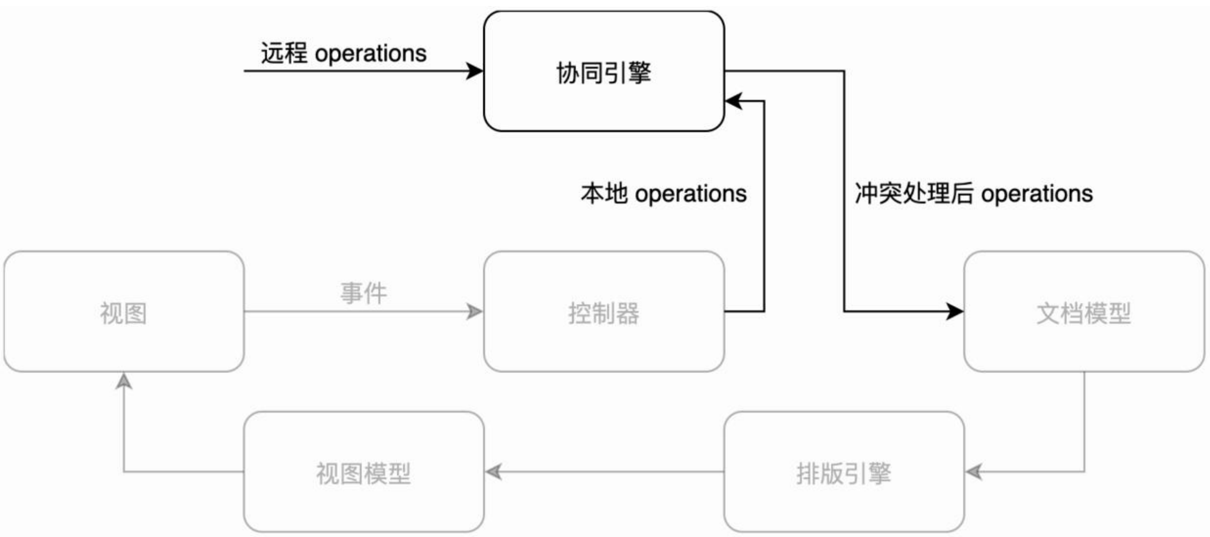
编辑器支持 OT 算法的关键是需要支持 operation 的编辑和应用。

在钉钉文档内，用户的一切编辑行为，都转换为 9 种原子 operation 组合，并使用 operation 驱动文档模型更新，而非直接修改文档模型。因此只需要针对数据流做简单的调整，即可支持协同编辑。



4、编辑器接入协同能力

在上图的数据流基础上加入协同引擎，接收本地及远程 operations 后，使用 OT 算法处理冲突。由此解答前文中视图要设计成数据驱动视图的特性，由于协同是基于数据进行而不是视图，使视图感知冲突处理的唯一方式就是让数据驱动视图。



5、Conflict-free Replicated Data Type(CRDT)

业内主流协同算法除了 OT，还有 CRDT，钉钉文档在一些小功能中也开始试点，比如点赞、已读未读等，是由 CRDT 来实现冲突处理。



五、开源计划



钉钉文档富文本编辑器已经沉淀为 SDK 并赋能内部 30+产品。接下来会推进开源计划，可以关注钉钉文档知乎专栏或加入协同文档钉钉讨论群。

钉钉文档知乎专栏

协同文档技术讨论群

该二维码7天内(12月19日前)有效，重新进入将更新

CRDT 实时协作技术在稿定编辑器中的应用

摘要：多人实时协作能力对许多 Web 生产力应用至关重要，但其始终存在较高的技术门槛。本次分享将介绍稿定在平面设计编辑器业务中基于 CRDT 库 Yjs 的实时协作功能落地实践，由此展望近年崛起的 CRDT 去中心化协作架构的可能性。

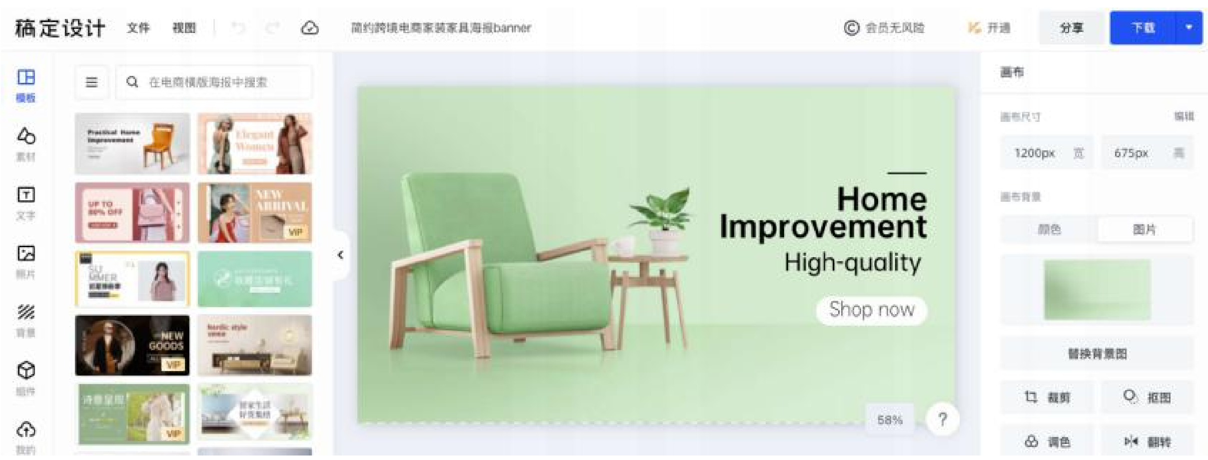
分享人：王译峰(雪碧)，Web 开发者，毕业于中国科技大学，现供职于稿定科技前端团队，先后负责了平面设计编辑器、3D 文字、跨端图形框架等项目的研发。其同样活跃在技术社区，在知乎和掘金的博客与技术回答阅读量累计超百万，译有《JavaScript 二十年》。

一、背景知识

1、业务背景

稿定平面设计编辑器：每月服务数万合作设计师与上百万用户的生产力工具。

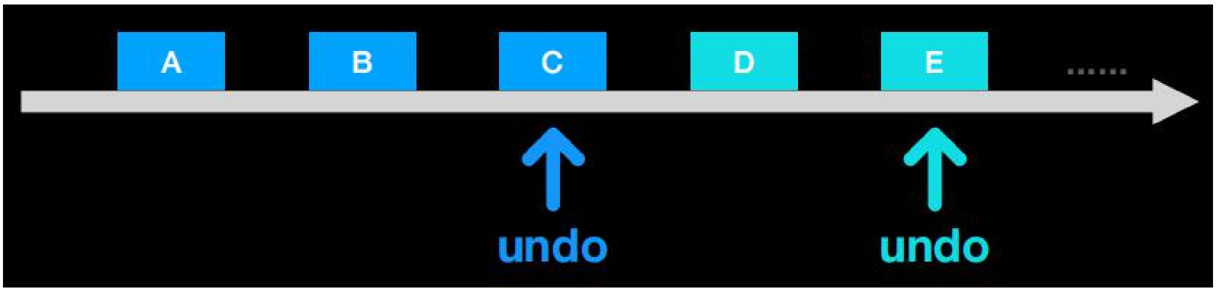
- 区分 C 端个人版与 B 端企业版，后者支持团队协作；
- 支持富文本编辑，但主要数据结构仍为树形，元素均为具备较多属性的节点；



2、从状态管理到实时协作：前置知识概述

- 常规「状态管理」在软件工程研究中的抽象：Event Sourcing 模式
  - 将 model 操作变为可序列化的 operation 数据，如：Redux；
  - 构建 operation 栈，在 undo/redo 时移动栈指针应用 operation 即可；

- 实时协作场景下的挑战
  - 由于网络延迟，分布式产生的 operation 在各客户端可能以不同顺序合并；
  - 由于用户仅可撤销自身改动，比如下图 A-E 五个状态，第一个用户编辑 ABC 第二个用户编辑 DE，第一个用户要撤销一定是将 C 状态撤销而不是 E，故需要能任意“移除”operation 栈底的某项；



3、如何使朴素的 Event Souring 机制适应协作场景？

- 引入类似 git 的 rebase 机制
  - 服务端唯一确定 operation 顺序，在各客户端 rebase 后应用；
  - 需依赖中心节点，亦不便支持离线编辑；

- 引入对 operation 数据的变换（Operational Transform，即 OT）
  - 将 operation 在服务端变换，客户端只需应用变换后的 operation；
  - 需引入中心节点，有较多经典案例但工作量大；

- 引入特殊的 model 数据结构（Conflict-free replicated data type，即 CRDT）
  - 每个用户具备唯一 UID，其每次操作均有 version vector（类似逻辑时间戳）；
  - 每个对象字段均关联 version vector，合并时依逻辑时序 last win 即可。

二、CRDT 库 Yjs 的接入实践

1、CRDT 库 Yjs 概述

- 源于 YATA 算法，较 RGA 等业界原有 CRDT 算法获得了显著改进；
- 最坏时间复杂度为合并远程 insert 时的 O(N^2)，其余场景均不超过 O(logN)；
- 社区已为其实现 Slate、Quill、ProseMirror 等编辑器的 binding 插件；

CRDT	LOCAL		REMOTE	
	INS	DEL	INS	DEL
WooT	$O(H^3)$	$O(H)$	$O(H^3)$	$O(H)$
WooTO	$O(H^2)$	$O(H)$	$O(H^2)$	$O(H)$
Logoot	$O(H)$	$O(1)$	$O(H \cdot log(H))$	$O(H \cdot log(H))$
RGA	$O(H)$	$O(H)$	$O(H)$	$O(log(H))$
YATA	$O(log(H))$	$O(log(H))$	$O(H^2)$	$O(log(H))$

Table 1: Worst case time-complexity analysis, adapted from [1].

2、Yjs 接入实践

- 其 API 类似 model 层，提供 YMap、YArray、YText 等 Shared Type；
- YModel 较难完全透明地与常规 model 同步生命周期，故建议维护 binding 层；
- YDoc 可在各客户端实例化，并关联 WebsocketProvider 透明地同步状态；

```
import * as Y from 'yjs'

const yDoc = new Y.Doc()
const yElements = yDoc.getArray('elements')

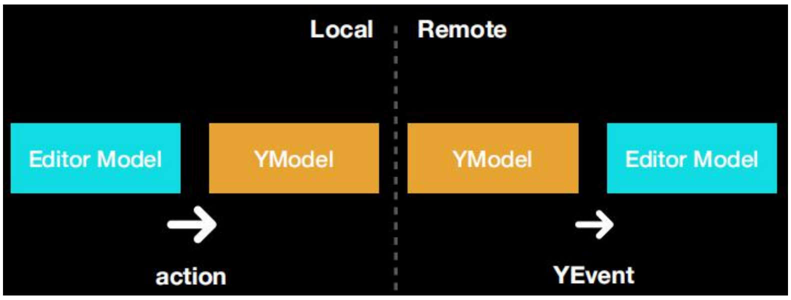
const yElement = new Y.Map()
yElement.set('left', 100)
yElement.set('top', 100)
yElement.set('type', 'image') // 可使用非 YText 的字符串
yElement.get('left') // 注意此时还拿不到结果

yElements.push([yElement]) // 关联到 YDoc 节点下
yElement.get('left') // 状态已可获取，并在 YDoc 实例之间共享
```

Yjs 基本使用方式示例

3、状态双向同步方案

- 为编辑器建模 action；
- 各类 action 无需可序列化，相当于简易 operation；
- binding 层根据 action 同步修改 YModel 即可；
- 本地→远程:基于编辑器 action 修改 YModel，见下面代码图中 commit；
- 远程→本地:基于 YEvent 修改编辑器 model，见下面代码图中 remoteObserver；




```
let isRemoteUpdating = false

async function remoteObserver(events) {
  isRemoteUpdating = true
  // 根据远程更改的 YEvent 修改编辑器 model
  // ...
  isRemoteUpdating = false
}

function commit(action) {
  if (isRemoteUpdating) return

  // 忽略本地更改触发的 YEvent，避免死循环
  yStore.unobserveDeep(remoteObserver)

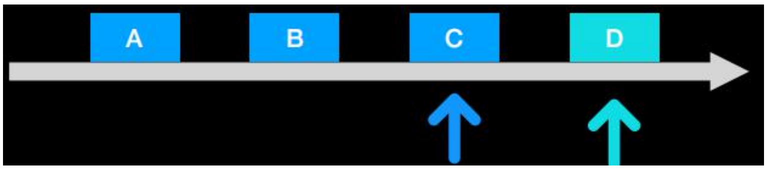
  // 在此根据编辑器 action 修改 YModel
  // ...

  yStore.observeDeep(remoteObserver)
}
```

4、历史记录

- 核心设计准则
  - 用户只能撤销自己的改动；
  - 用户从状态 A 独立撤销 N 次之后再重做 N 次，要能回到 A；

- Y.UndoManager 符合上述准则
  - 为每个用户维护一个本地的 undo 栈；
  - 对 undo 前后两做 model 状态做 diff，将该 diff 以 YEvent 形式发送；



如图，独立撤销 C，理想中应获得 A B D ，但实践中应获得 A B C D C’ ，且只需保证 C C’等价于空操作，即可保证 undo 后，redo 的 A B C D C’ C 仍为 A B C D。

5、后端配套方案

- CRDT 无需中心节点，大幅降低了后端开发成本；
- 配套后端方案的关键点：
  - 基于 Yjs 配套的 y-websocket 搭建房间，自动广播房间内各客户端改动；
  - 可将 YEvent 的 Uint8Array 更新数据缓存至 Redis，令服务端实例横向扩展；
  - 在 Node 端对 YModel 做序列化操作，即可保存文档版本；
  - 可参考社区库 Hocuspocos 封装供鉴权、房间新建等业务使用的生命周期钩子。

三、状态同步方案优化

1、面向嵌套节点树的状态同步优化

- 朴素实现
  - 一对一映射 YModel 结构（YModel + YArray 表达力与 JSON 等价）；
  - 更新时以元素为粒度整体更新节点；

- 改进
  - 支持树形结构扁平化与 fractional-indexing；
  - 支持字段级的 diff 增量更新。

2、树形结构扁平化

- 问题
  - 朴素实现会将元素成组操作建模为“先删除子元素，再整体添加新组”；
  - 对“A 成组 -> B 解组 -> A 撤销 -> B 撤销”路径，会出现重复元素；

- 优化方案
  - 将 YModel 建模为扁平的 Map<UUID, YModel>结构；
  - 为元素维护 parentId 和 index 字段，在 reparent 操作时更改 parentId 即可；
  - 优化后可支持元素的 reparent 与 shift 语义。

3、Fractional-indexing

- 问题：扁平化后需维护元素 index 字段，更新范围大且可能在撤销后出现重复；
- 优化方案
  - A B C 元素下标不再为 [0, 1, 2] 形式，而是 { A: 0.25, B: 0.5, C: 0.75 } 结构；
  - 在 B C 之间插入 D 时，获得 { A: 0.25, B: 0.5, C: 0.75, D: 0.625 };

```
import {generateKeyBetween} from 'fractional-indexing';

const first = generateKeyBetween(null, null); // 'a0'
// Insert after 1st
const second = generateKeyBetween(first, null); // 'a1'
// Insert after 2nd
const third = generateKeyBetween(second, null); // 'a2'
// Insert before 1st
const zeroth = generateKeyBetween(null, first); // 'Zz'
// Insert in between 2nd and 3rd. Midpoint
const secondAndHalf = generateKeyBetween(second, third); // 'a1V'
```

Node 中 fractional-indexing 演示

4、字段级 diff 增量更新

- 问题

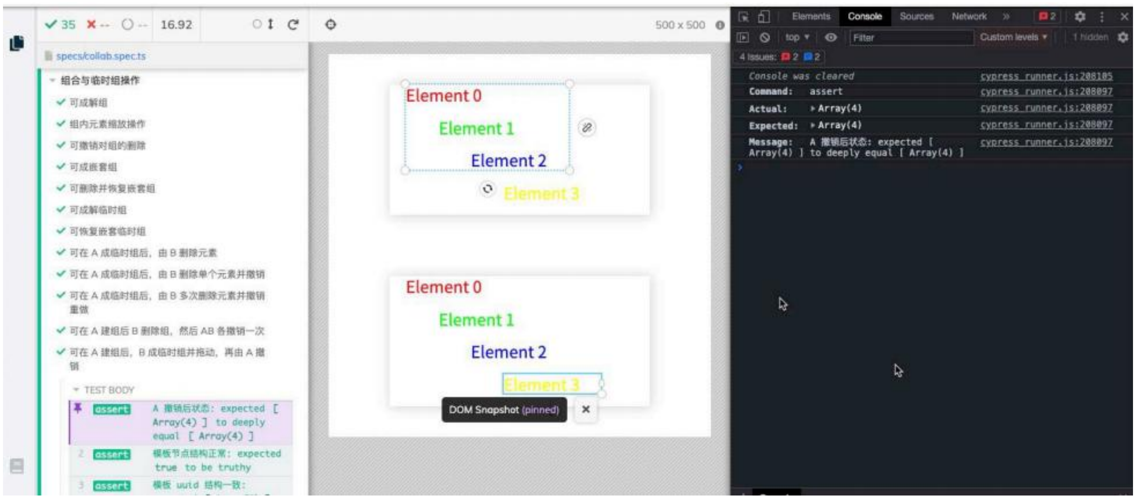
  - 若每次同步均全量更新 YModel 字段，每次 undo 时的粒度也只能精确到元素；
  - 对“A 更改颜色 -> B 更改文字 -> A 撤销”路径，将额外撤销 B 的改动；

- 优化方案
  - 精确到元素字段级的自动 diff；
  - 仍提交 change_element 类型的 action，附带上更新完成后的元素实例；
  - 对比 action 中的元素新状态与相应 YModel 的旧状态，仅更新 diff 变化的字段；

四、自动化测试与 benchmark

1、自动化测试与 benchmark

- 开发环境：基于 Cypress + Vite 的 TDD 环境；
- 性能测试：基于 Node 的 CLI 压测工具；



2、自动化测试：面向协作场景的测试工具

- 基于 Cypress Component Testing 搭建本地调试页：
- 支持完全可视化的 DOM 环境，无需 Node 侧模拟 JSDOM；
- 支持 Vite，具备无等待的开发体验；
- 支持 mount 单个 Vue 组件实例，较传统的 visit URL 形式 E2E 显著更轻；

- 使用两个本地 Editor 实例，以交换二进制更新数据的形式通信，无需 WebSocket；
- 可对比两个实例的 model 状态与 parentId 等字段合法性，及时发现异常状态。

3、性能测试：使用 Node 模拟多用户场景

- 在 Node 端用 y-websocket 实例连入房间，模拟常见编辑器 YModel 更新；
- 内存占用较高，2M 模板 100 客户端每 500ms 局部更新 5 分钟，需 3GB 内存；
- 房间人数增长后，新用户连入时有显著延迟（对当前 50 人量级足够使用）。



五、总结

- CRDT 方案已在前端实时协作领域具备强竞争力；
- CRDT 的设计需离散数学理论，但工程落地所需的仍是基础数据结构知识；
- 技术方案应具备理论可靠性，否则正确实现的方案仍可能有重大问题。

虚拟偶像诞生记 — 数字人行业和技术探究

我们理解的数字人



近几年数字人的概念在互联网中异常火热，业界也出现了很多热门虚拟数字人或虚拟偶像，比如 Lil Miquela，洛天依、初音、鹿鸣、暖暖、AYAYI 等。这些数字人的商业模式或市场价值通常是：**通过运营积攒流量，依靠流量变现**。例如：

- 品牌代言（IP、事件）
- 粉丝经济（二次元）
- 虚拟主播（游戏、带货）

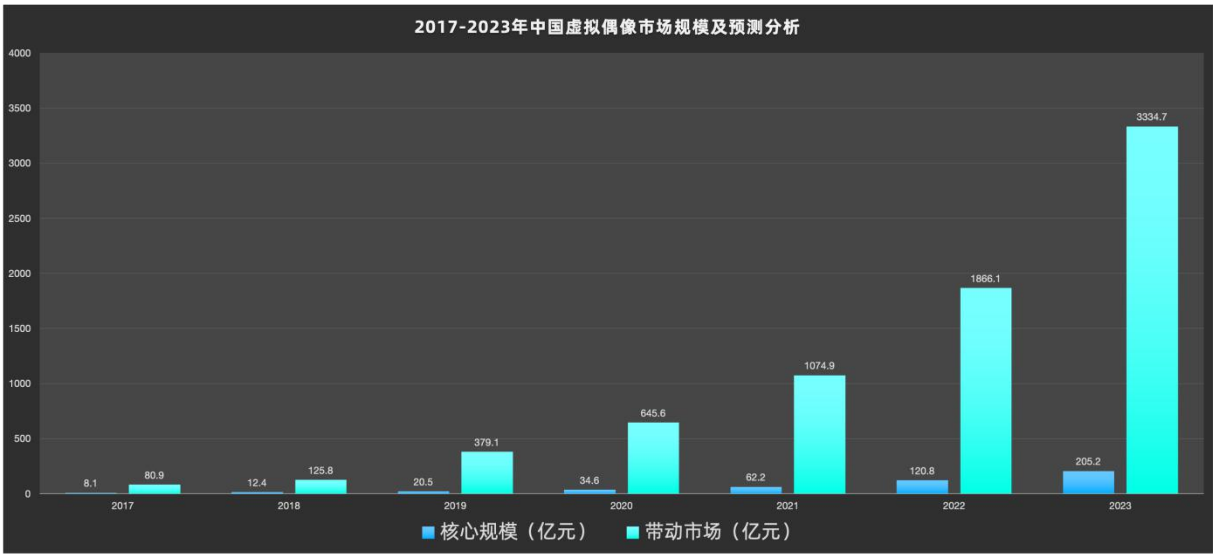
本文为接下来讲到的数字人，定义了三个核心要素：

一、**形** - 具备人或拟人的外观，具有特定的相貌等人物特征。
二、**动** - 拥有和人相似的行为，具有用语言、表情和肢体等表达的能力。
三、**神** - 拥有比拟人的思想，具有识别外界环境、并能与人交流互动的能力。

这三条要素对于虚拟数字人的“完成度”来说是一个递进的关系。

市场现状

近几年，虚拟数字人在电商、金融、影视、游戏和金融等行业都拥有不同大小的市场规模。例如：**虚拟偶像行业 2020 年中国的市场规模为 34.6 亿元，预计 2021 年将达到 62.2 亿元。**

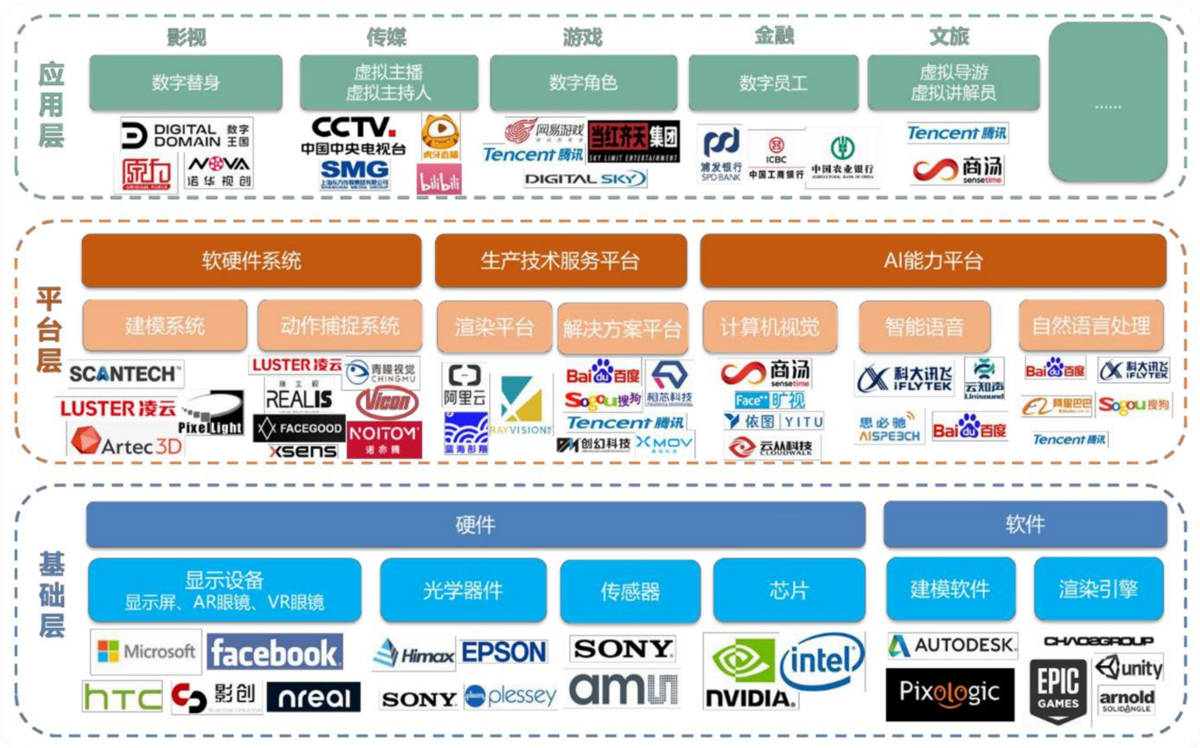


市场规模增长，也印证了技术的发展：制作成本逐年降低、形象外貌更加真实、语言交流更加自然。从出现虚拟数字人至今，已经渡过了 3 个重要的时期：

- **初创期：市场初见增长，技术不确定性，入行门槛高。**
- **发展期：市场竞争者增多，技术逐渐成型，入行门槛低。**
- **平台期：市场红海到来，平台化成熟，领头羊 + 小众。**

解决方案

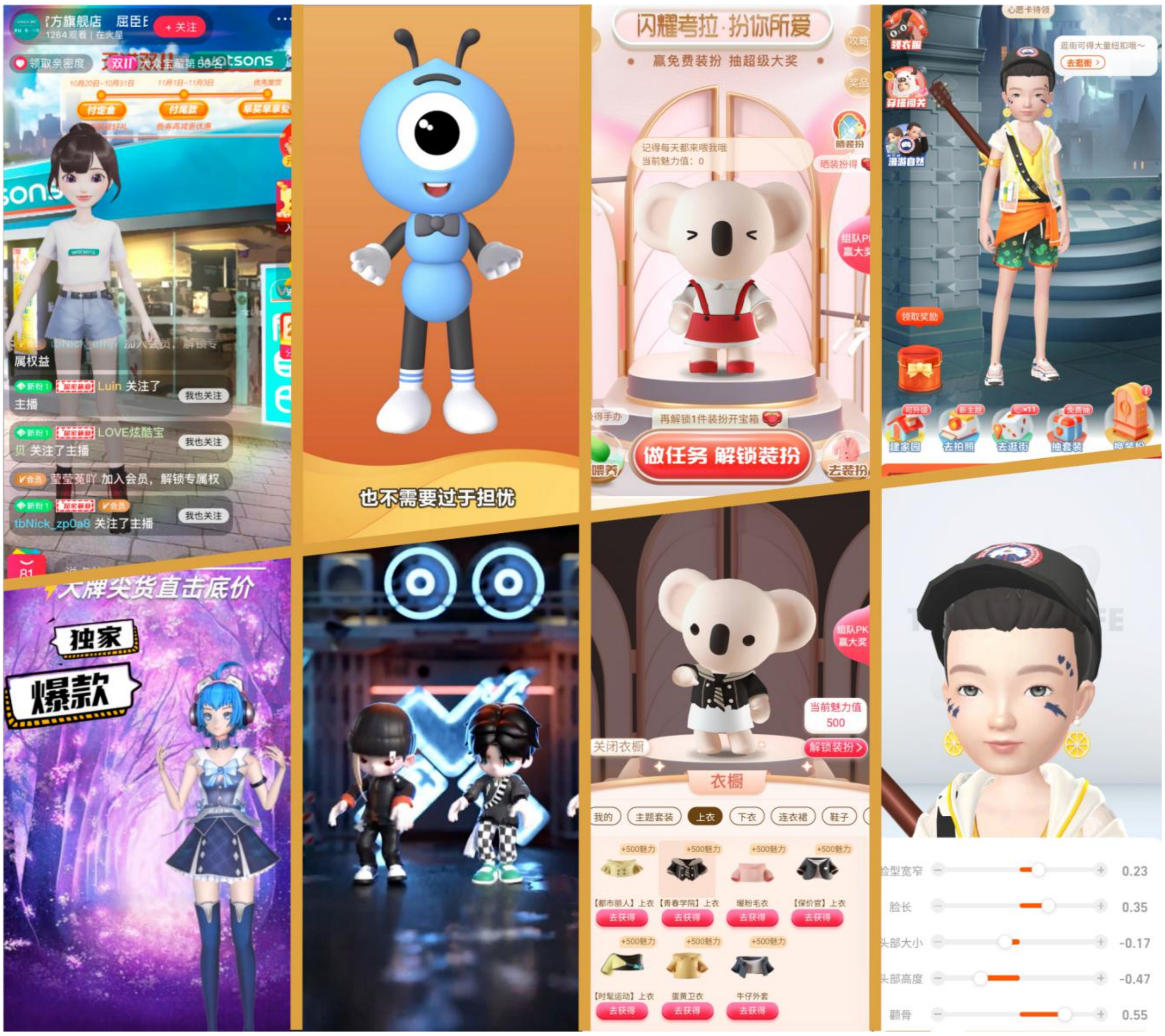
在如今的平台时期中，各个厂商也聚集在一起提供不同层次的解决方案：



- 基础层：为虚拟数字人提供基础软硬件支撑，硬件包括**显示设备、光学器件、传感器、芯片等，基础软件包括建模软件、渲染引擎**。 仅有少数几个顶尖科技公司才具备优秀的软硬件实力。
- 平台层：包括软**硬件系统、生产技术服务平台、AI 能力平台**，为虚拟形象的制作及开发提供技术能力。提供平台服务的企业非常多，通过出售服务和技术来服务其他企业。
- 应用层：除了最终的企业用户，一些具备出色营销和运营能力的公司或团队也为这个行业带来了不错的想法和创意。

我们在做什么

在今年年初，在阿里前端委员会的互动图形方向中成立了**虚拟角色小组**，这个小组有这几只团队组成：大淘宝互动团队、达摩院智慧数字人团队、优酷数字人制播团队、考拉互动&内容导购团队、蚂蚁数字与金融内容社区团队。他们一起分享和研究虚拟数字人领域的技术和应用。业务涉及**游戏、视频、直播**三个主要的场景。



- 游戏：虚拟数字人在游戏行业基本是标配，非常多的游戏都会需要塑造人物，其中能让玩家通过自定义塑造人物形象功能的这一类游戏，自定义塑造人物形象的功能也称为**“捏人”**。
  - 淘宝人生：这是手淘 App 内一款可以打造虚拟形象的游戏，包含了捏脸、穿搭、美颜、拍照等功能，逛街、家园等玩法。
  - 养考拉：这是考拉海购 App 内的一款可以养成考拉的游戏，包含了装扮功能，喂养等玩法。
  - 它们均采用 Web 为主的技术方案，通过自研的引擎完成渲染、表情、动作等人物的形和行。
- 视频：虚拟数字人的短视频可以给用户带来感官上的优秀体验，还能给业务带来增量的收益。**在制作虚拟数字人的视频时，可以通过动作捕捉、智能识别、导演系统等方案，让虚拟数字人栩栩如生**。
- 直播：直播和虚拟数字人的结合仍然处于早期的探索阶段，因为两种现象级的产业需要结合且形成新的或增量的商业模式并不容易。**这其中涉及到的技术也有实时动作捕捉、算法训练合成、在直播场景下的云渲染推流等等**。

我们一起来创作

随着应用场景专业和深入，技术研究也会涵盖工程和算法等多种综合性的解决方案，且在不同的应用场景下，侧重点也会不太一样。接下来以大淘宝互动团队的淘宝人生业务为例子，分别从美术生产、渲染风格、捏脸换装、表情动作、导演系统和语音合成六大主题为大家呈现如何打造超级虚拟偶像。

形

这个章节我们来完成虚拟数字人的形 - “具备人或拟人的外观，具有特定的相貌等人物特征”。







从模子中刻出来 - 美术工作流

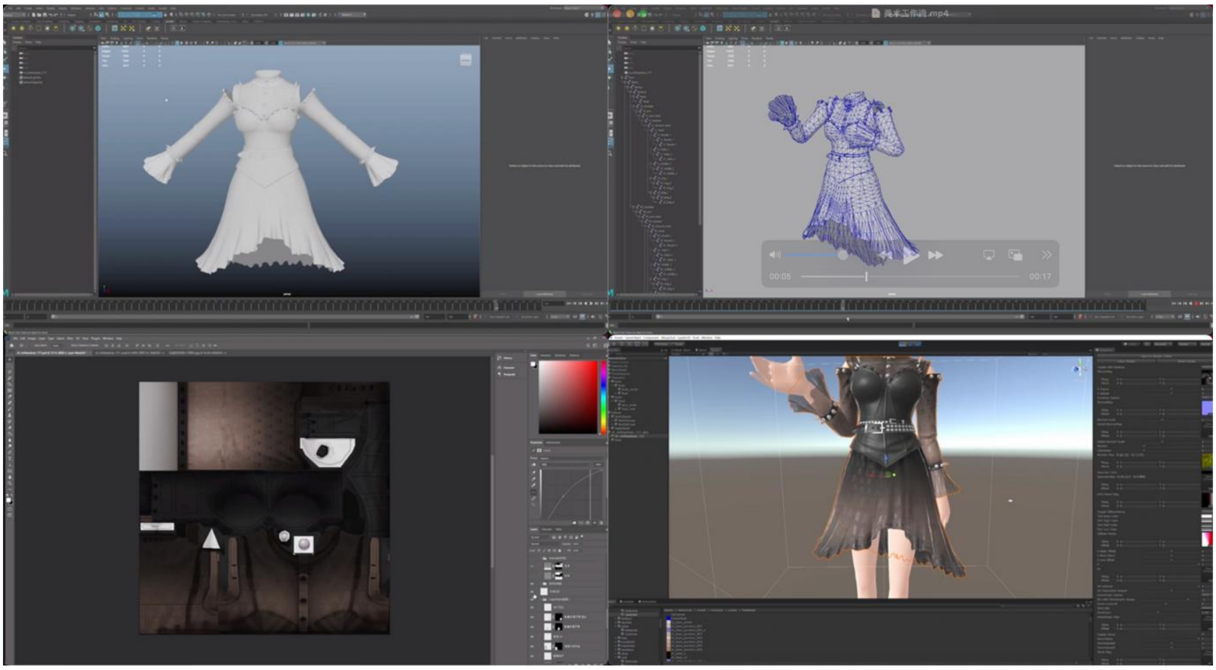
我们最先需要确定虚拟数字人的**基础形体特征**，比如使用真人比例的 7 头身，还是卡通比例的 5 头身；男性，女性，还是动物拟人等。基础形体特征确定后，可以交由 3D 美术来生产基础形体。整个过程，通常是在传统 DCC 软件里完成，但 3D 美术和 2D 美术最大的不同是，**在 2D 内容的美术生产中，技术可以在美术产物交付后再介入；而在 3D 内容的美术生产中，技术就需要直接介入生产过程中**。原因是：3D 内容的美术生产流程相对较长也较复杂，美术和技术之间需要频繁的合作才能保障交付质量或交付效率。我们也称作这个过程为 **3D 美术工作流**。

举个形象例子：艺术家开始设计并雕刻一个杯子模具，生产线需要解决模具用什么材料、如何将材料注入到模具中、以及如何顺利脱模等“技术性问题”。通常这些就是由技术来负责，并

需要提前和艺术家协商制作模具的规范才能确保后续的工程顺利进行，最终完成模具的交付。3D 美术工作流也有很多大同小异的方案，**大同是因为 3D 内容生产也是有一定的工业标准的，小异是因为细节会和艺术家使用的软件、技术实现的引擎有密切的关系**。以淘宝人生中的美术

工作流举例大致经历这几个步骤：

- 用 Maya 制作白模，骨骼，**把过程产物暂存在 OSS，并提供预览工具**
- 用 Photoshop 制作贴图，**把贴图上传至 CDN**
- 在 Maya 里**定制 GLTF Exporter** 插件，导出 glTF（包含模型数据、骨骼数据、材质、贴图数据）
- **Web 端内嵌材质编辑器进行自研材质的效果调整**
- 在 **EVA Figure 引擎**通过 GLTF Importer 导入人体的 glTF，并**定制材质 Shader** 进行渲染。

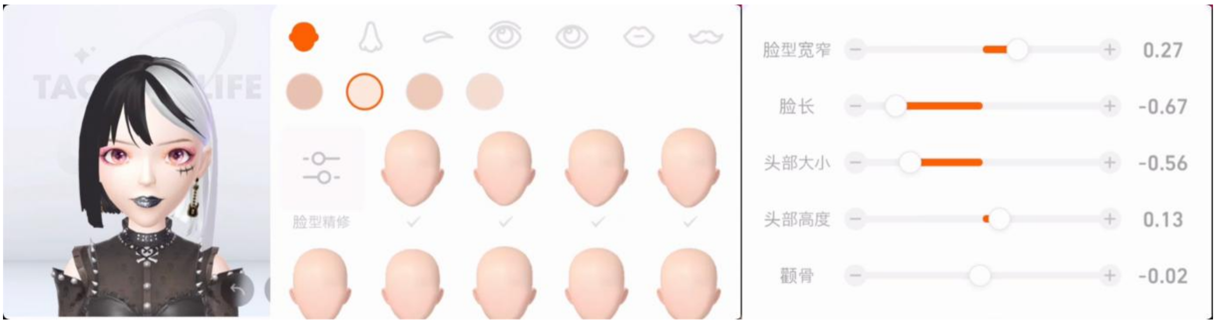


经过前期和艺术家的磨合，最终会形成一套针对当前需求的美术工作流，并稳定运行下去。

精雕细琢 - 捏脸

完成了角色的基础形体后，我们每个人都可以用它来继续完成自己想要的那个样子。虽然不同

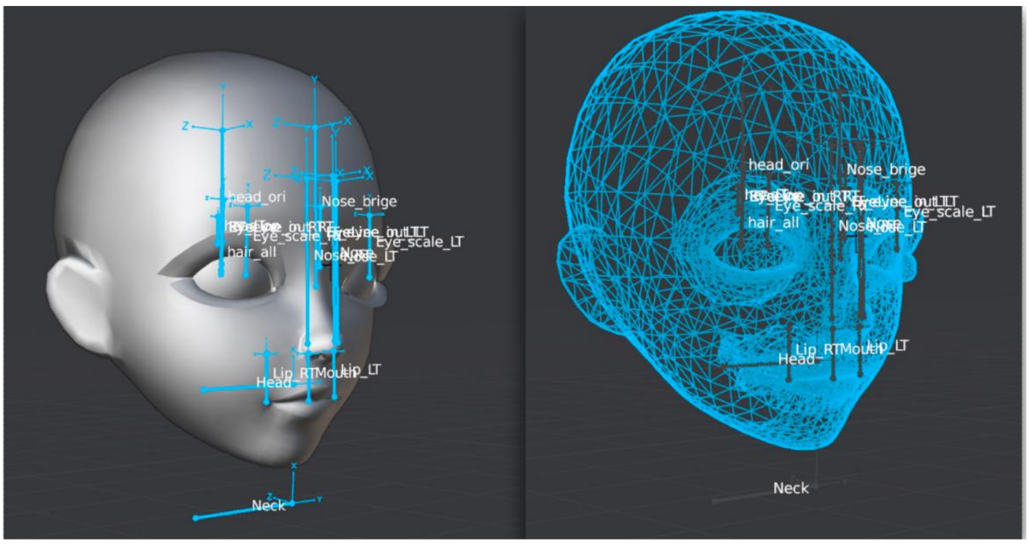
的长相在美术环节中也能完成，但成本相当的高，每个人喜欢的样子都不一样，那么就要很多次的美术生产或修改。所以我们为基础形体加入了捏脸系统，你就可以轻松的完成定制化的要求。淘宝人生提供了捏脸功能，可以体验到捏脸的乐趣。捏脸技术的的实现基础就是对已有的模型数据进行局部的修改和变化，最终达到千人千变的目的。



那是如何能通过这样的系统来达到改变基础模型的目的的呢？通常，一组模型数据就是顶点数据的集合，改变模型通常就是改变顶点的数据，而改变顶点数据又通常有两种方式：

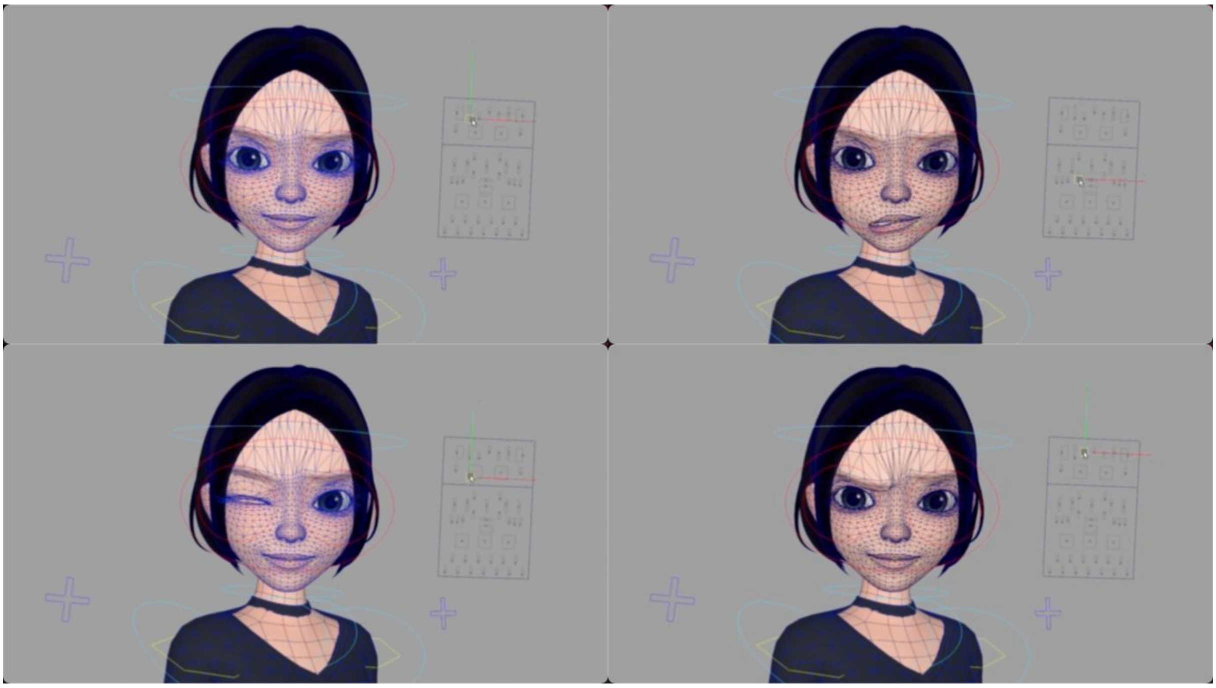
骨骼蒙皮

通过“外力”对顶点施加某种“变换”，简单来说就是包含了位移、旋转和缩放三种变换的一组数学公式。为了完成这种变换，这股外力可以使用“骨骼”来完成。这里说的骨骼同样可以理解成人体的骨骼，当手指上关节的作用发生运动变化时，手型就会发生变化。在捏脸功能中，在淘宝人生中，我们给脸部预置了大约 20 多种骨骼，可以改变头围、眼球、眼角、眼眶、颧骨、脸型等等。



混合变形

骨骼带来的顶点变换很粗矿，无法完成对嘴形状的定制，因为这样一个看似简单的外表，实际上在模型中会涉及数以万计的顶点进行不同规律的变换。于是，我们就为这一组顶点变换专门设置变形器，业界一般称为“**Morph Target**”**或**“**Blend Shape**”。这种变换的原理是给顶点准备一个基准位置，再提供一个”极端变化“后的最大位置，之后乘以一定的”权重比例“，就能让这个顶点在基准位置和极端位置中的任一位置。但因为脸部需要局部变形的地方太多，且一个变形可能涉及数以万计的顶点，所以对实时计算压力也是不小的。

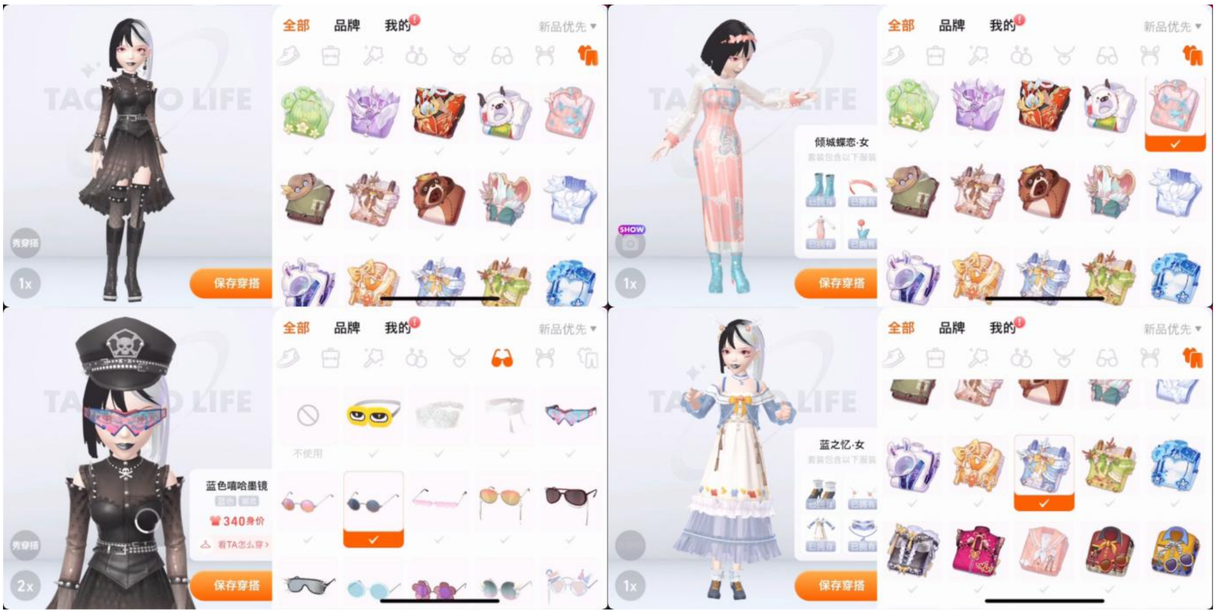


两种技术比较来说，骨骼蒙皮**简单高效但不够灵活，而顶点变形更加自由但制作和计算成本都较高**。所以在实际开发中，哪些用骨骼，哪些用顶点变形，这是一个**权衡“效果”与“效率”的工作**，需要具体去看，反复拿捏。**在淘宝人生中，”捏脸“功能小部分使用了骨骼蒙皮，大部分使用了混合变形。这也是长期磨合和经验累积的结果**。

时尚穿搭 - 换装和美颜

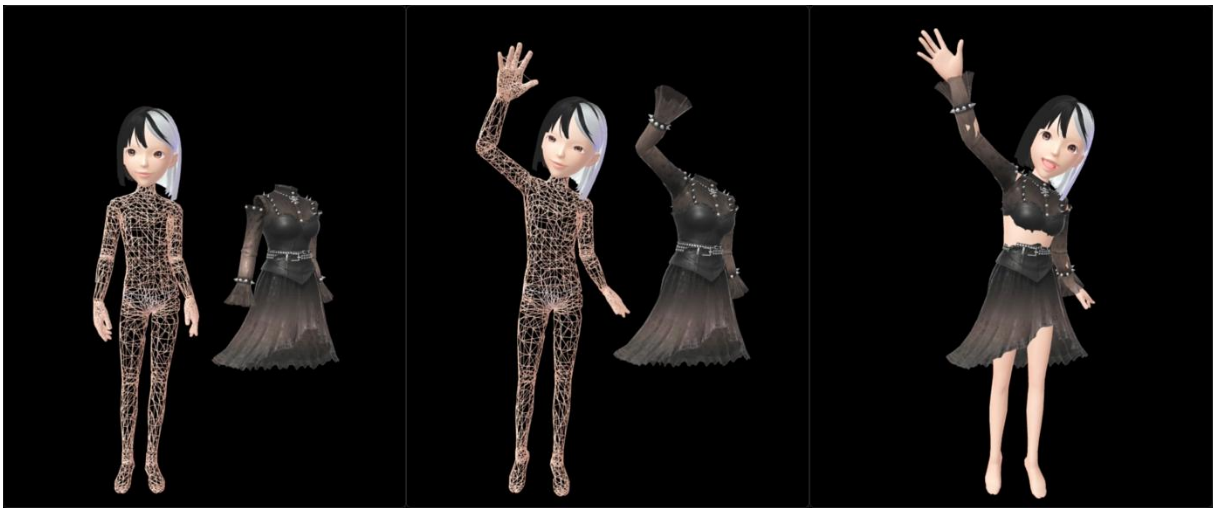
有了基础形体和长相后，我们必须得给角色穿上时尚的衣服，画上美丽的妆容。对于人类来说，穿衣和画妆是个“动作”，在虚拟世界中，我们仍然要能很好的完成“穿”和“画”这两件重要的事情。

换装

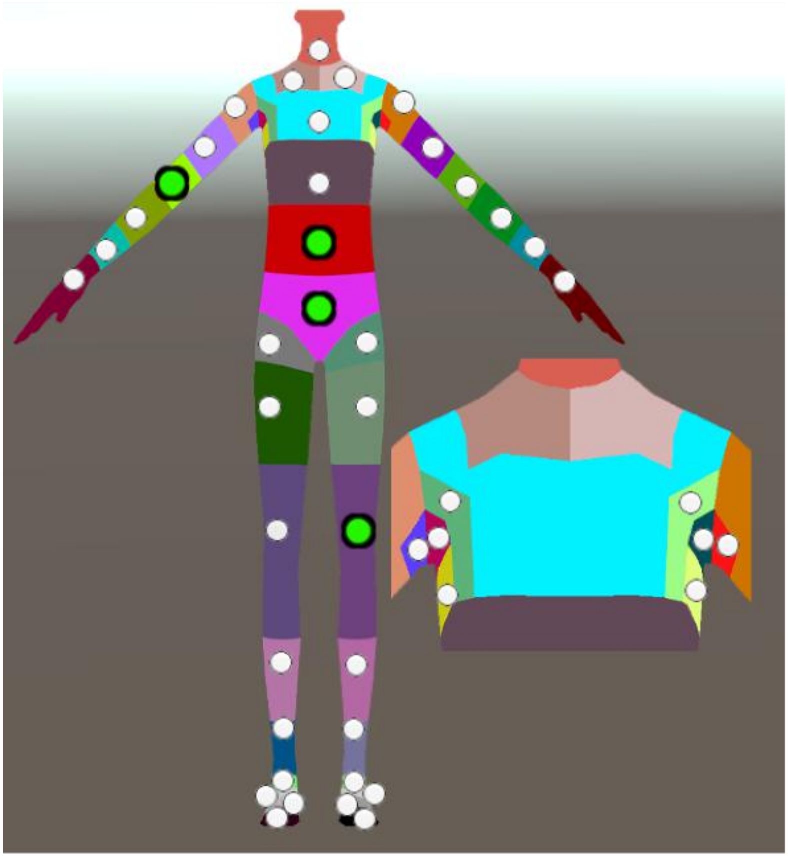


现实生活中，衣服穿在身上它就和皮肤紧贴着或者有一定空隙，这种想法放到虚拟世界中其实非常难实现。因为皮肤和衣服其实就是 Mesh（网格），当衣服穿在身上时实际上是两组网格“碰撞”在一起，于是会引出如下两个问题：

如何在身体做动作时，衣服也跟着身体“做动作”。身体是拥有骨骼的，骨骼外包裹着一层“皮肤”，用同样思路，衣服实际上也是包裹在同样骨骼上的一层“皮肤”。在淘宝人生中，我们将身体和衣服采用同一套骨骼模板，并在渲染时实现了两份骨骼数据的实时“同步”。



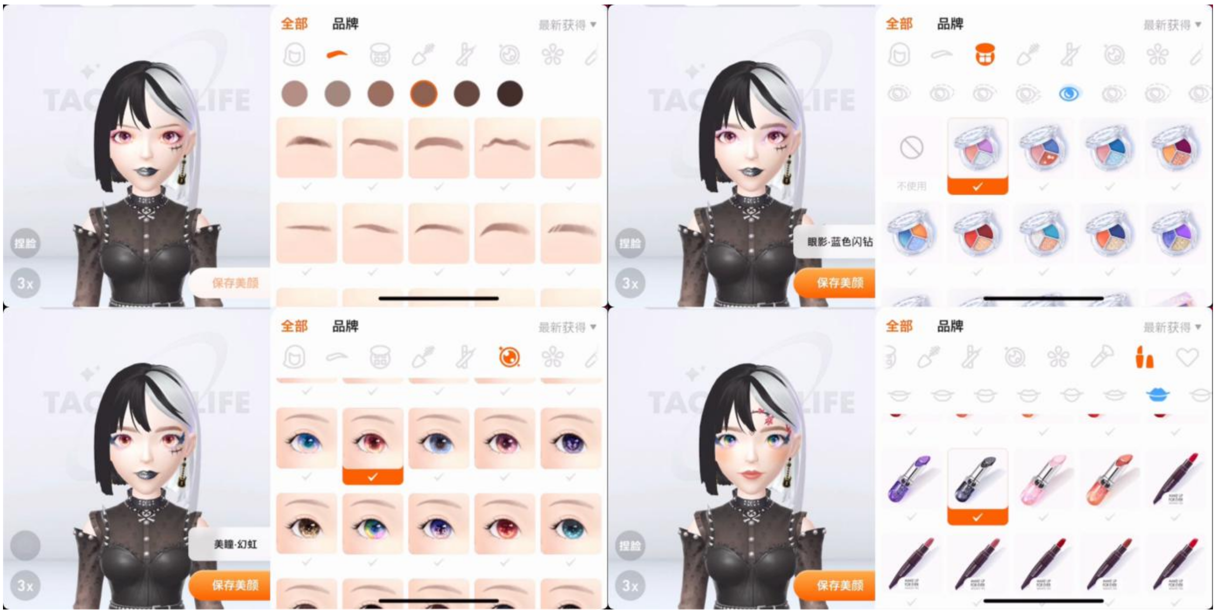
如何解决身体的网格穿透到衣服外。用相同骨骼的方案解决衣服“穿”在身上的效果非常巧妙，但也容易出现问题，比如某件衣服局部非常内凹，那么就很容易出现身体的皮肤突出在衣服之外的问题，俗称“穿模”。因为仔细调整道具成本实在太高了，所以我们也做了个取巧的方案：**通过把人体进行“切割”，并对每件衣服遮挡的人体部位进行标记，当渲染某件衣服时，直接隐藏被遮挡部位 Mesh 即可。**





有了这两项技术的攻坚后，我们通过和美术生产约定相关的规范和工作流程，就可以批量生产衣服，而换装仅仅只是加载不同的模型而已，无须逐个特殊处理即可达到目的。

美颜



妆容的细节要求非常多，所以最便捷的方式就是使用贴图。基础形体的脸部已经有一张基础贴图了，可以简单理解成“素颜”；要在素颜上画不同的妆容，我们做法是“动态合成纹理”。整个过程分为两步。

1、-渲染到纹理（Render To Texture）：第一步创建一个可渲染对象（Render Target），把基础纹理贴图渲染到这个可渲染对象上，再把妆容纹理贴图渲染到同一个可渲染对象上。这一步需要注意艺术家在绘制妆容纹理时，需要和基础纹理的 UV 是一一对应的。

2、使用可渲染纹理：第二步就是把合成好的纹理渲染到模型上。



二次元还是三次元 - 渲染风格

最后到了给它设定风格的时候。有人会喜欢真实系，有人也会喜欢卡通系，也会有人喜欢朋克风，也会有人喜欢纯欲风，这些不同的风格需要依靠渲染功能来完成。当我们说到渲染时，都会提到图形渲染管线，它可以根据不同的需求做很多组合和调整，比如最简单的渲染管线是：加载模型 ---> 顶点着色 ---> 片元着色 ---> 光栅化。其中**片元着色**的步骤，就是用来完成材质纹理的绘制使之达到想要的角色风格。对材质纹理的绘制通常分为两大类：

PBR

全称 Physically Based Rendering，顾名思义是基于物理的渲染，既然是基于物理的，那么它**渲染的最终结果会非常接近我们的真实世界**。所以也就很好理解，这类材质可以决定人物是**写实或超写实风格**的。此技术由 8 大核心理论和几个重要的光照模型组成，在此不一一列举，有兴趣的同学可以去阅读《Real-Time Rendering》中的 PBR 相关章节，或是 SIGGRAPH 的《Physically Based Shading in Theory and Practice》系列文章。例如在淘宝人生中，根据不同部位选取渐变贴图上采样色的方式模拟了次表面散射的技术，实现了人脸皮肤的红润通透感。

NPR

全称 Non-photorealistic Rendering，即非真实感绘制。其最主要的应用之一，就是非常盛行的二次元风格，特别是日式卡通风格。和 PBR 不一样，NPR 不会去追求各种物理学模拟，而是从油画、素描、卡通动画中得到启发和灵感。比如，经常被使用的人物描边、卡通着色、边缘光、头发高光等。这些特殊的材质渲染也都能在找到专业的论文和例子，大家可以自行搜索。



动

这个章节我们来完成虚拟数字人的动 - “拥有和人相似的行为，具有用语言、表情和肢体等表达的能力”。





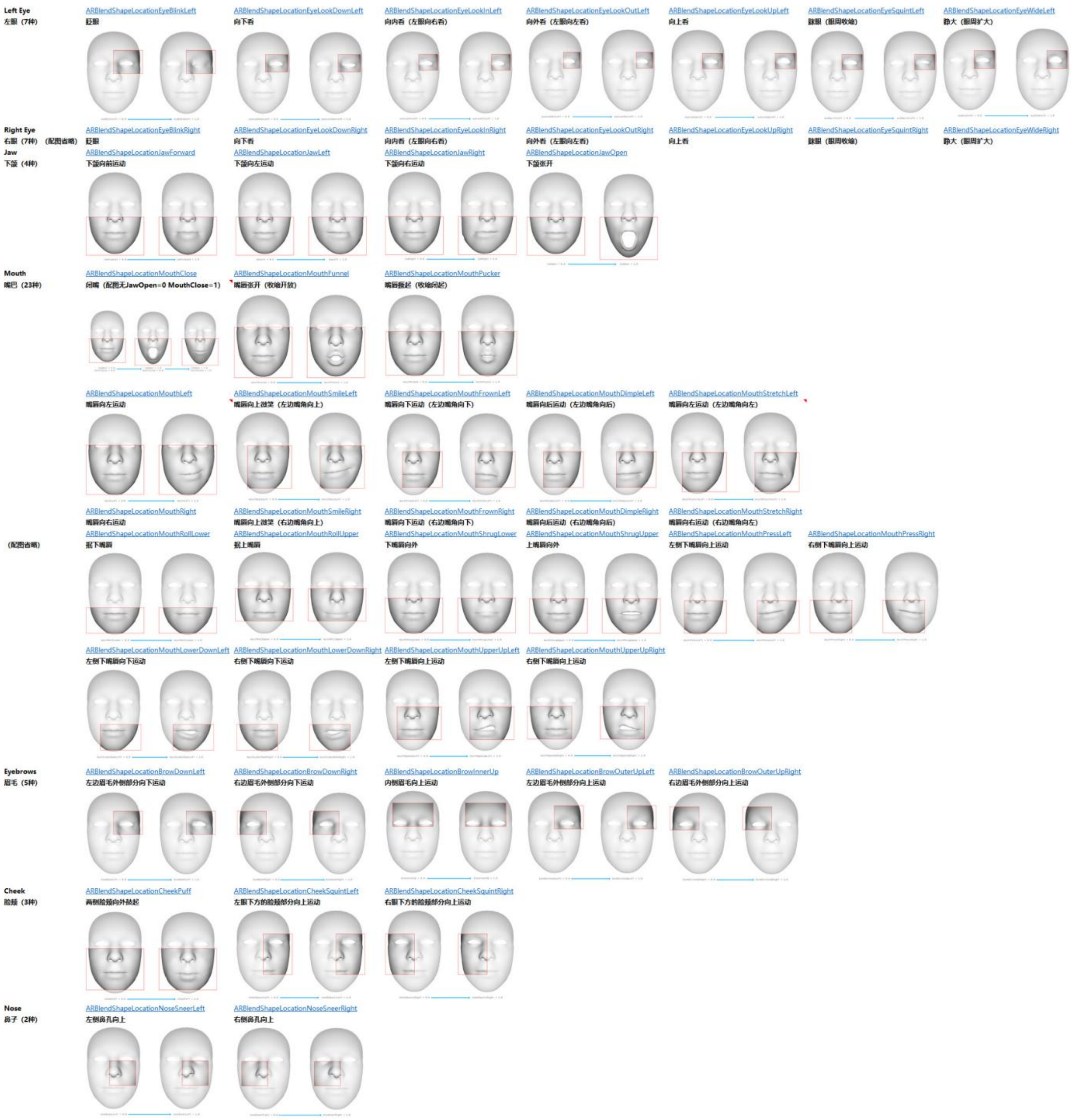
表情和动作

成功塑造虚拟数字人的关键之一是真实且细腻的表情和动作。真人的表情和动作是依靠骨骼和肌肉相互作用而呈现的，在虚拟世界中，我们同样用数字技术来模拟这些骨骼和肌肉的作用，呈现表情和动作。在上面“形”的章节里，已经提到了脸部使用了骨骼蒙皮和混合变形的方式来改变顶点，在“动”的章节里，仍然会使用这两项技术，并且通过动画的方式让顶点“动”起来，从而完成相应的表情和动作。

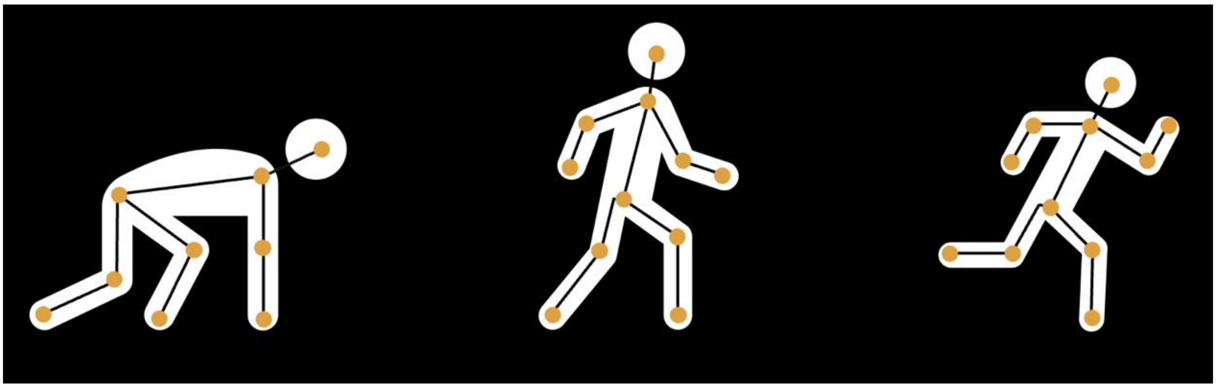


手 K 动画

在脸部表情的刻画中，**顶点动画（也就是 Morph Target Animation）是主要实现技术之一**。在淘宝人生的“捏脸”功能中骨骼蒙皮是用来确定脸的大小和五官的位置的。混合变形则是用来完成五官包括脸颊、额头部位的变形，其中用于表情动画的 BlendShape 多达 50 个，它和苹果的 Animoji 使用的 BlendShape 非常接近，这也是业界公认的可以做到绝大多数表情（当然很多微表情非常难达到）的设定。

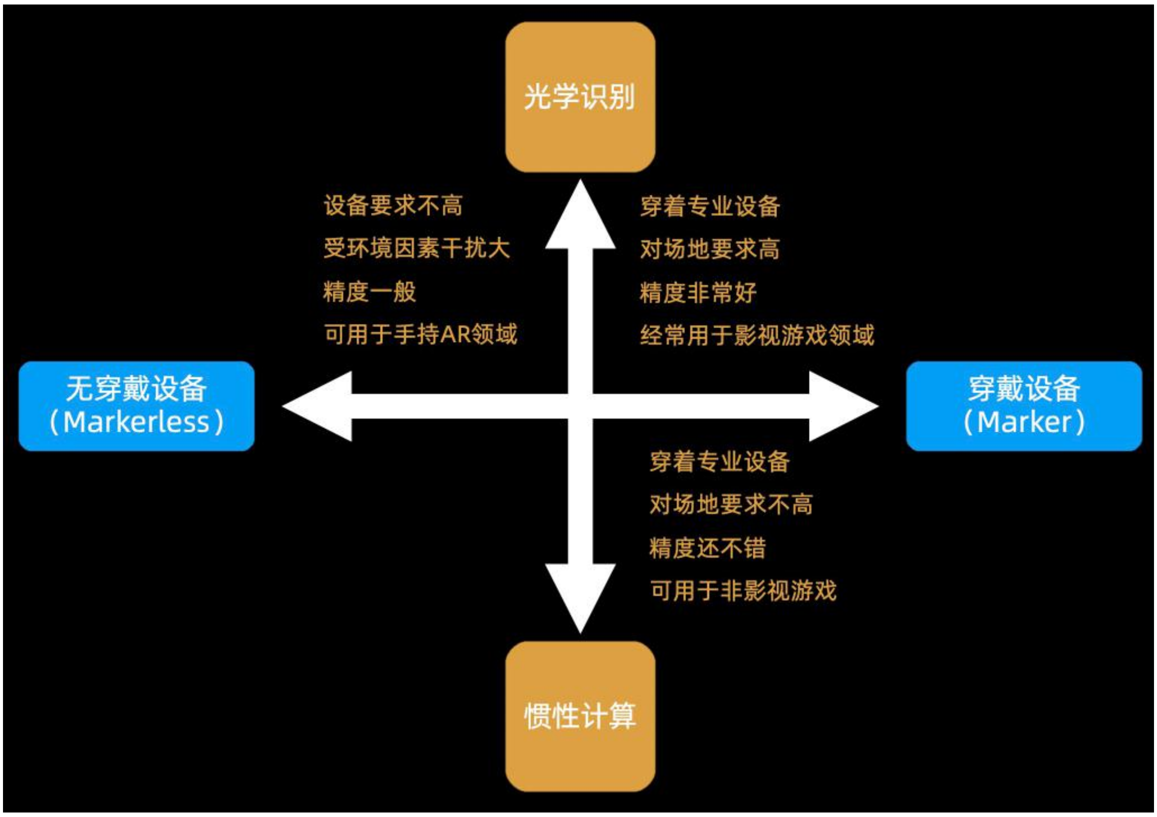


在肢体动作的制作中，**骨骼动画（也就是 Skeletal Animation）是主要实现技术之一**。骨骼动画包含了**骨骼（Skeleton）和蒙皮(Skin)两种数据**。首先把 Mesh 的顶点绑定在骨骼上生成蒙皮，这些顶点会受到多个不同权重骨骼的影响，然后通过改变骨骼的朝向和位置来生成动画，这时蒙皮就会跟着骨骼一起动起来。



动作捕捉

制作动画的成本相对都比较高，原因是上述两种动画技术通常是**关键帧动画**。假设一个人抬起手肘，那么手臂和手腕也会因此而产生连贯性的动作。如果要实现这样一段动画，关键帧将会非常之多，制作起来也会异常困难。为了有效降低制作成本，我们需要用到动作捕捉的技术，通常我们把动作捕捉技术分为 2 大方向 4 个类别，可以用四象限图来描述：



我们用手机摄像头完成的 AR 互动游戏、AR 面具等功能实际就是光学识别+无穿戴设备这一象限内的技术。

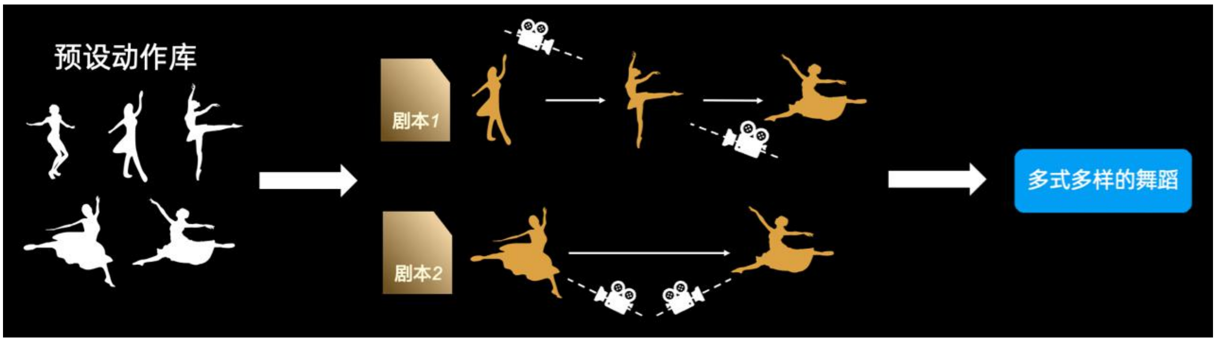


编排 - 导演系统

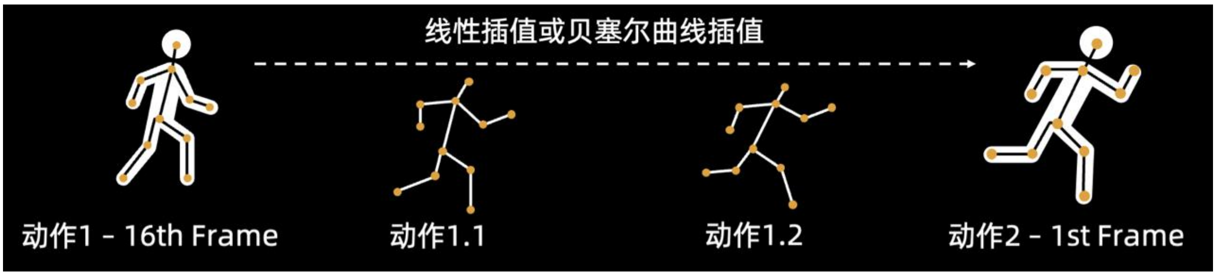
我们再来对比下几种方式制作的成本和灵活度。

偶像跳舞的制作	成本	灵活度
手K动画制作	单次成本高	高
动作捕捉制作	启动成本高、单次成本适中	中
预设动作编排	启动成本高、单次成本极低	较差

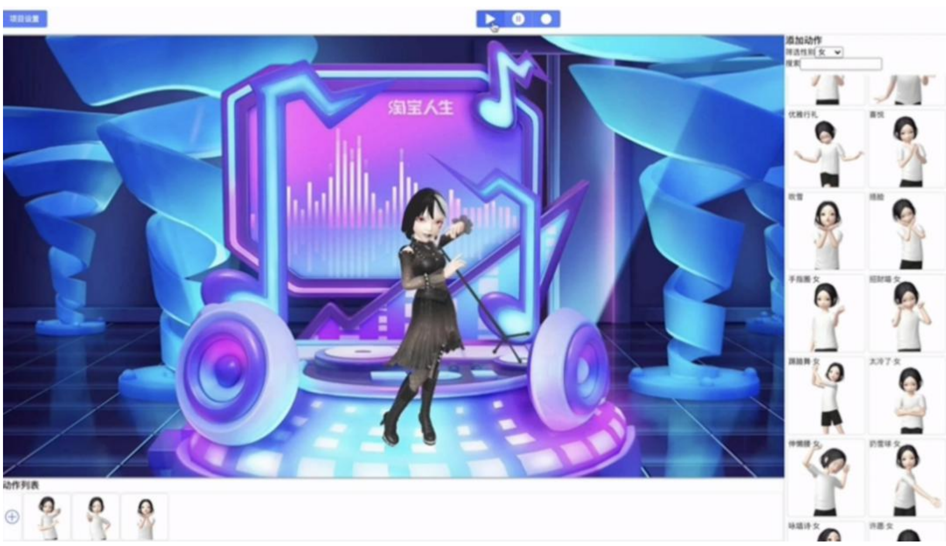
手 K 动画的制作成本无疑是最高的，灵活度和效果也是最好的，同时也要求经验丰富的绑定师和动画师才能完成。动作捕捉的制作需要一套专业的动捕设备，一个能容纳这些设备和能满足做所需动作的场地，一个动作制作周期少则几个小时，多则几天。在面临诸如众多舞蹈场景的运营需求下，最适合的方案是把多个已经制作完成的动作，自由或智能编排成一个剧本，就像拍戏时剧本是导演提前设计好的，演员只需依照剧本表演出来就行，故也称为“导演系统”。



动作的串联需要解决一个问题：动作 1 到动作 2 如何过渡衔接。这便需要用到**混合动画（Blending Animation）技术**。混合动画的基础原理是**以动作 1 当前状态为起点、动作 2 的指定状态为终点，计算关键帧的插值**。简单的可以使用**线性插值**，复杂的也可以使用**贝塞尔曲线插值等**。关于混合动画的知识点和方案非常多，也适用于不同的场景和需求，大家可以自行搜索，例如 Unity 和 Unreal 也都提供了很多不同的混合方案。



有了导演系统后，就可以提供自由的组合能力，比如想举行一场虚拟演唱会，也可以通过导演系统完成表演。



神

这个章节我们来探索虚拟数字人的神 - “拥有比拟人的思想，具有识别外界环境、并能与人交流互动的能力”。我们在对“神”的研究上，仍然处于非常初级的阶段，一方面它需要大数据的支撑，另一方面它对于前端这个岗位来说已经有相当的距离了。为了让虚拟数字人更加真实，“神”会是未来重点的研究方向。

性格流露 - 自然语音合成

虚拟数字人的语言表达能力需要用到语音合成技术，比如 **TTS（Text To Speech）**。阿里达摩院有着非常完善的 TTS 引擎，用它就能让虚拟数字人开口说话。不过这也仅仅是开口说话，大家其实都能感受到这样的语言非常苍白无力，毫无“情感”，无法在不同性格和情感时表达出来的不同语气。在业界可以看到一些优秀的成果：YAMAHA 的歌声合成系统“VOCALOID”（初音未来和洛天依均在使用），谷歌基于深度学习的端到端语音合成系统“Tacotron”，以及讯飞的语音合成系统等。阿里达摩院也在持续研究更加符合自然表达的语音合成系统，通过为虚拟数字人设定性格，并借助大数据的深度学习来产生风格迥异的情感型语气。大家可以细细评委下以下三段真人、普通 TTS 和风格化 TTS 的效果。

【此处插入音频】原声.wav
【此处插入音频】普通 tts.wav
【此处插入音频】风格化 tts.wav

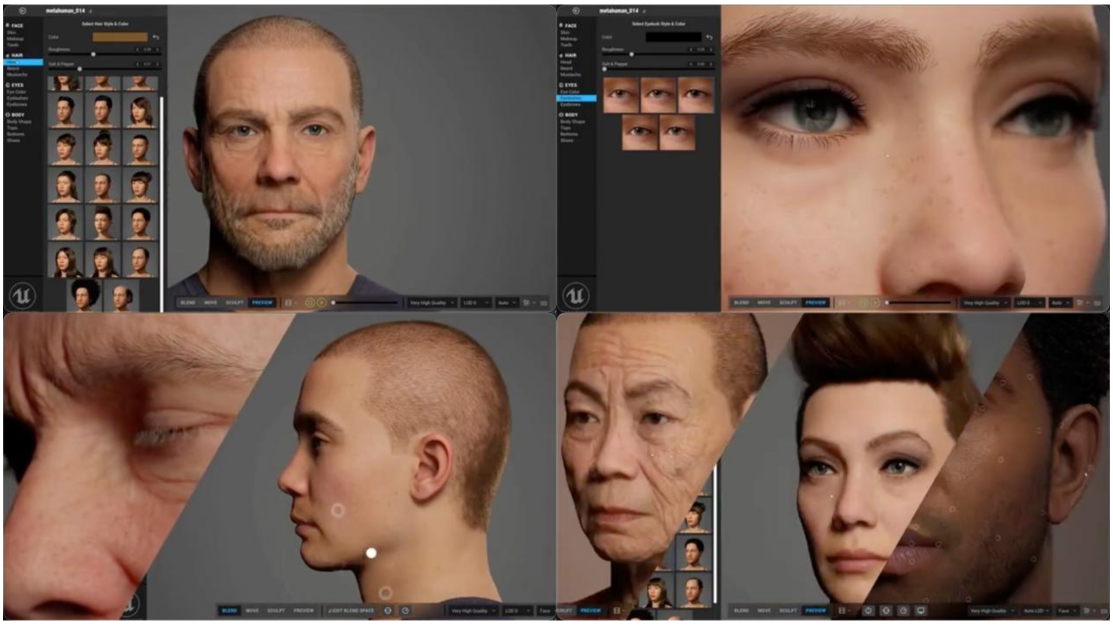
至此，在现有的技术下我们完成了一个超级虚拟偶像的登场。



我们向往的数字人

淘宝人生是基于 Web 技术打造的数字人，在经历了 2 年的技术打磨和升级改造，我们也遇到了 Web 数字人技术的困境。在性能方面，Web 应用和原生应用存在无可忽视的差距。WebGL（基于 OpenGL ES 2.0）作为 Web 主要的图形接口在 Vulkan、DirectX、Metal 面前无论是能力还是性能上都无法追赶。再加上不同的移动设备在软硬件上仍会出现奇奇怪怪的兼容性问题。这一系列困境已经成为了 Web 中的数字人一个很难跨越的天花板。

反观行业中的数字人技术，超写实渲染、微表情、肌肉模拟、物理材质、光线追踪等等，让身处 Web 技术中的我们望尘莫及。同时，阿里的虚拟数字人技术也才刚刚起步，其中基础软硬件、中台技术、大数据支撑等也都起步较晚，这些都给我们带来了不小的困难和阻力。



面对这些困境和差距，我们针对淘宝人生中虚拟数字人技术的发展也会从各个方向上去做努力和尝试。

首先是基于 Web 技术的优化：在 Serverless 渲染云服务基础上，在 EVA Figure（虚拟人像渲染引擎）和 Puppeteer 技术的结合下，借助最新的 WebGPU/WASM 等，让虚拟数字人的渲染效果和质量得到提升。我们也在积极同阿里云云服务团队、大淘宝 Node 架构团队一起合作建设 Web 技术下的云渲染流程。计划把它应用在一些非实时的渲染任务上，例如生产淘宝人生用户形象的全身照片、短视频、动作帧图等等。这些产物也能被淘宝人生业务或其他业务所使用。其次是对业务能力的升级，在业务和技术相互滋养的土壤中，不断将业务能力积累成平台服务，为阿里的虚拟数字人技术贡献点点滴滴的经验，并将部分解决方案通过商业化上云的方式服务大众。

然后在元宇宙、超写实、XR/6G、脑机接口等等行业风向标的辐射下，畅想未来 Web 数字人技术可能性。

文末特别感谢下阿里前端委员会图形互动虚拟角色小组成员的出色工作，让我能完成此文章内容。也欢迎大家持续关注虚拟角色小组和各个团队的成果。

参考文献

- 《艾媒咨询|2021 中国虚拟偶像行业发展及网民调查研究报告》https://www.iimedia.cn/c400/79469.html
- 《三维软件知多少》http://www.bgteach.com/article/40
- 《三维文件格式知多少》https://www.bgteach.com/article/132
- 《glTFTutorial》https://github.com/KhronosGroup/glTF-Tutorials/tree/master/gltfTutorial
- 《Vertex Transformation》https://www.khronos.org/opengl/wiki/Vertex_Transformation
- 《WebGL Skinning》https://webglfundamentals.org/webgl/lessons/webgl-skinning.html
- 《MorphTarget》https://en.wikipedia.org/wiki/Morph_target_animation
- 《Real-time Rendering》https://www.realtimerendering.com/
- 《MovingFrostbite toPBR3.0》https://seblagarde.files.wordpress.com/2015/07/course_notes_moving_frostbite_to_pbr_v32.pdf
- 《StylizedRenderinginGame》http://stylized.realtimerendering.com/
- 《基于物理的渲染（PBR）白皮书》https://zhuanlan.zhihu.com/p/53086060
- 《ARkitFaceBlendshapes》https://arkit-face-blendshapes.com/
- 《OpenGLSkeletalAnimation》https://www.khronos.org/opengl/wiki/Skeletal_Animation
- 《类卡通效果与写实人脸的 52 个 blendshape 效果对比及变化说明》https://www.bilibili.com/read/cv11763241
- 《MotionCapture》https://en.wikipedia.org/wiki/Motion_capture
- 《MotionCaptureSystem》https://scholar.google.com/scholar?q=Motion+capture+system&hl=zh-CN&as_sdt=0&as_vis=1&oi=scholart
- 《GameEngineArchitecture3rd-AnimationBlending》https://www.gameenginebook.com/toc.html
- 《游戏引擎动画系统阅读笔记》https://zhuanlan.zhihu.com/p/374343190
- 《语音识别技术》https://scholar.google.com/scholar?hl=zh-CN&as_sdt=0%2C5&as_vis=1&q=%E8%AF%AD%E9%9F%B3%E8%AF%86%E5%88%AB%E6%8A%80%E6%9C%AF&btnG=
- 《语音合成技术》https://scholar.google.com/scholar?q=%E8%AF%AD%E9%9F%B3%E5%90%88%E6%88%90%E6%8A%80%E6%9C%AF&hl=zh-CN&as_sdt=0&as_vis=1&oi=scholart
- 《MetaHuman》https://www.unrealengine.com/en-US/digital-humans

Web 端短视频编辑器的设计与实现 — 像做 PPT 一样做视频

摘要：相比图文，我们都喜欢视频内容，而在阿里小蜜中存在大量的图文知识，有一些问题很难用文字描述清楚，也有用户对阅读长图文内容反感或者理解困难，导致用户对答案的满意度提升困难，如果能够将图文知识转换为视频知识，对满意度的提升肯定会有助益。对于视频的生产，一般的方案是交由专业机构去创作，但这将花费很多预算，如果我们能提供一个工具，基于知识的通用结构沉淀一些视频模版，让用户快速创作出视频知识内容岂不美哉？让想法再奔放些，如果我们能直接从知识库中抽取结构化的知识内容直接生成视频或是半成品视频，用户只需要稍作调整就能发布，这想想就很酷吧？是的，小蜜视频创作工具我就是想做这样一件事情。

分享人：李志成(敦固)，来自阿里巴巴 CCO 一枚兴趣使然的前端工程师。

一、背景

1、我们为什么要做一个 Web 端短视频编辑器

阿里小蜜团队负责客服机器人项目，在业务上的需求是根据用户的问题，快速准确的给出答案，从而减少转人工率。

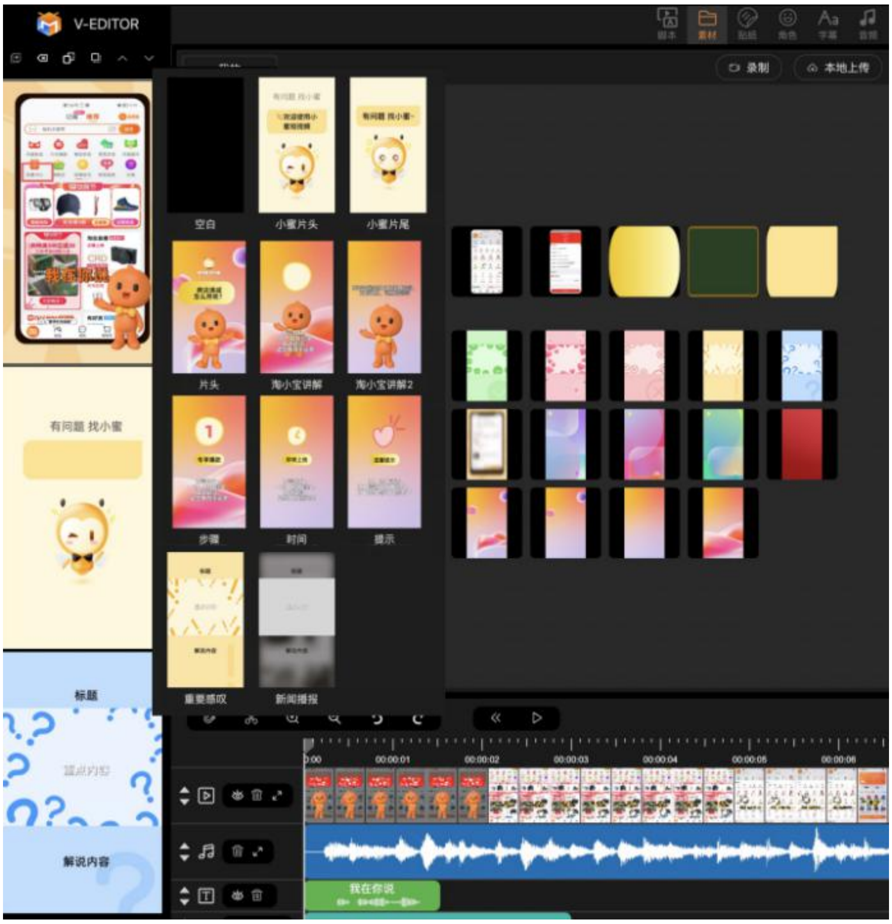
我们发现，图文形式看起来冗长乏味，同时因为文字过多而不易找到重点，尤其对老年人群不友好；相反以视频方式呈现则让内容变得有趣易接受，同时信息密度更高，也更适合老年人群。

当然，将内容视频化也存在制作成本高、制作效率低、专业难上手等问题。

2、如何让短视频制作像做 PPT 一样简单？

在设计初期的构思：

- 迁移 PPT 的使用经验，快速上手；
- 可定制的丰富模板应对不同场景；
- 智能脚本辅助创作；



而简单并且专业的短视频编辑器需要具备以下要素：

- 音视频编辑器的基本能力，包括：时间轴、字幕编排、动画贴纸、转场、滤镜、特效、TTS 配音、ASR 语音识别，等等；
- 纯前端实现；
- Web 端渲染。

二、编辑器设计

1、总体架构

如下图所示，编辑器总体架构分为三层，自上而下分别是：应用层、引擎层和依赖层。

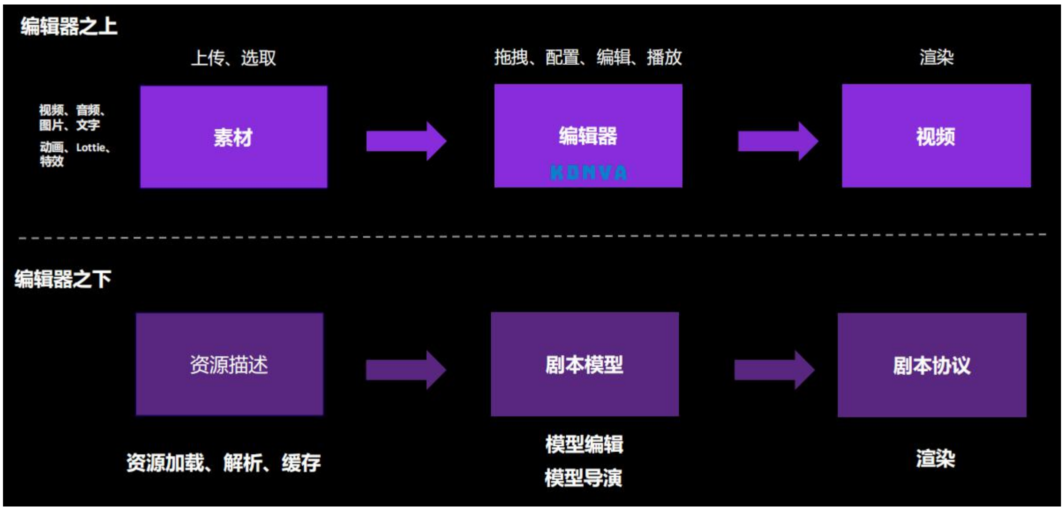


编辑器总体架构

- 应用层：可以在编辑器中看到的各种应用，包括素材库、模板库、智能脚本、剧本模型等等；
- 引擎层：包括资源管理器、导演引擎、舞台、渲染器以及服务；
- 依赖层：根据用途大概分成三类，视图、状态管理，动画、特效，音视频处理；

2、编辑器的设计

编辑器的设计首先从编辑器的输入输出考虑。如下图所示：

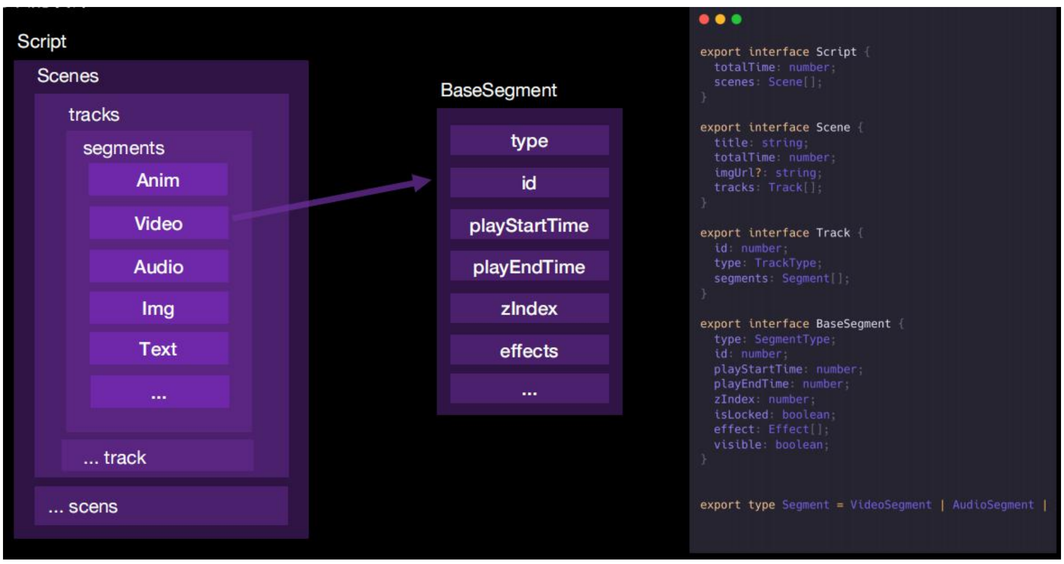


在编辑器之上，用户上传素材到编辑器进行拖拽、配置、编辑、播放等一系列操作，最后进行渲染输出视频，在这个过程中输入素材输出视频；

在编辑器之下，当用户上传素材对应得到素材的资源描述，并对素材进行加载、解析、缓存。当用户进行编辑时，对应下面的剧本模型编辑；当用户点击输出视频时，对应下面的剧本协议解析，并渲染生成动画。

3、剧本协议的设计

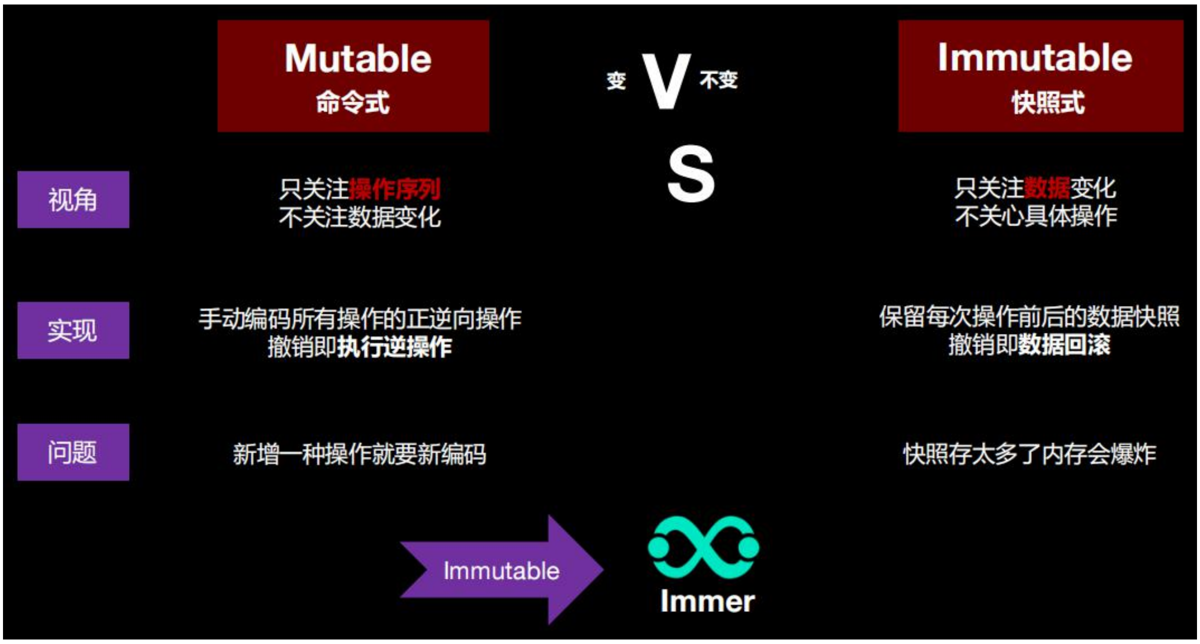
剧本协议是一个非常复杂的套娃结构。



4、以撤销重做功能为例谈变与不变的取舍

在编辑器设计中，选择采用可变数据结构，还是不可变数据结构，从下图的对比可以看出两种方式都各有利弊。鉴于前端会比较倾向函数编辑，最终选择快照式 Immutable 方式，使用 Immer 解决对数据结构的高效操作及编辑。

针对快照式的问题可以通过数据结构优化来解决，比如复用没有发生变化的节点，就可以有效节约内存。



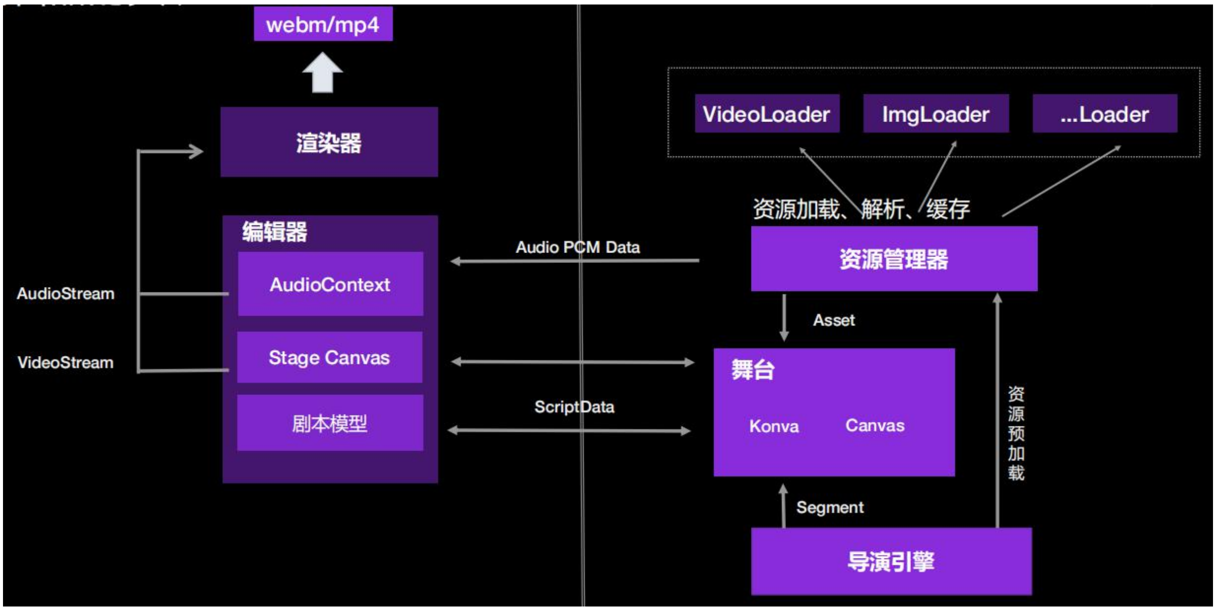
三、编辑器的实现

1、编辑器工作流

如下图所示：左边是渲染器和编辑器，由编辑器提供的 AudioStream 和 VideoStream 作为渲染器输入合成视频；

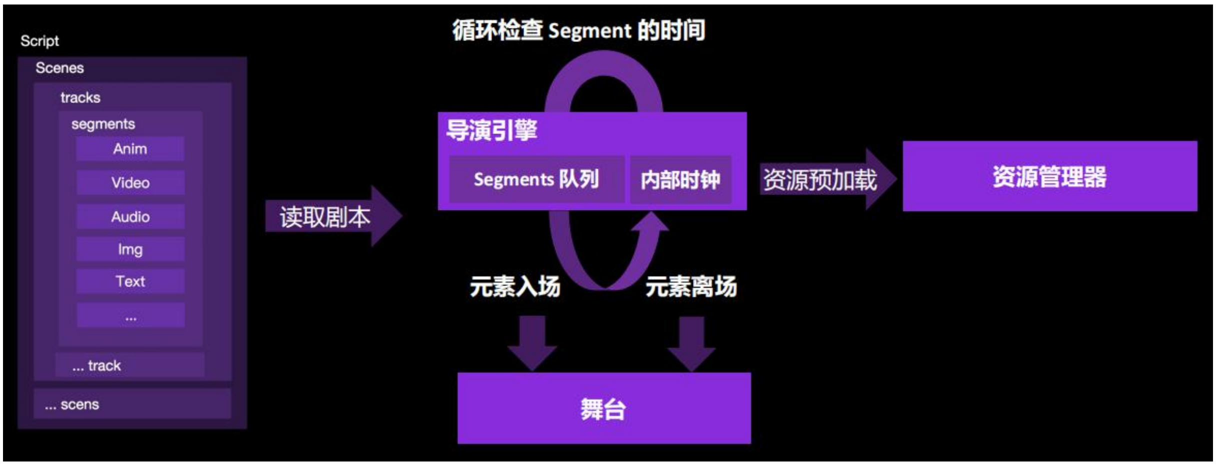
右边最上层是资源管理器，进行资源加载、解析和缓存，舞台基于 Konva 和 Canvas 供用户进行拖拽、配置、编辑等操作；

最下层是起核心主导作用的导演引擎，它将脚本模型发送至资源管理器进行资源预加载，然后支持整个舞台运作。



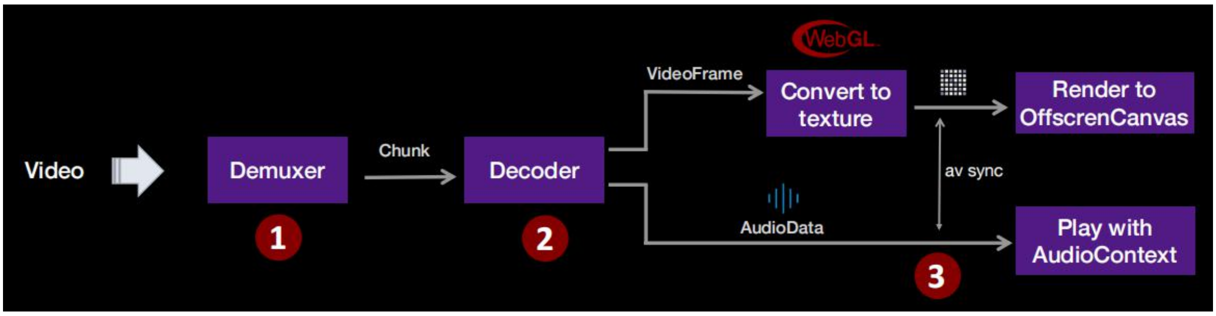
2、导演引擎

从输入输出的角度出发，导演引擎的输入是剧本，从剧本协议中读取剧本后将其中的元素进行排序，并加入 segments 队列，发送队列至资源管理器进行预加载。当用户点击播放时，内部时钟会不断触发检查元素的入场和离场时间并发送至舞台。这其中对内部时钟的精度要求非常高，尤其是音频，因此选择浏览器中的音频处理器 Web Audio API，它提供了理想的外部时钟和一致的时间序列，并通过 AudioScheduledSource Node 触发。



3、音频的处理与渲染

音频的处理和渲染有以下三个步骤（下图）：



1）解封装

把容器里压缩过的音视频流拆出来
音频：AAC/MP3/AC-3/...
视频：H.264/H.265/MPEG2/...

2）解码音视频码流

从压缩的编码中获得视频的像素数据和音频 PCM 数据
音频：PCM
像素：YUV/RGB

3）渲染与音视频同步

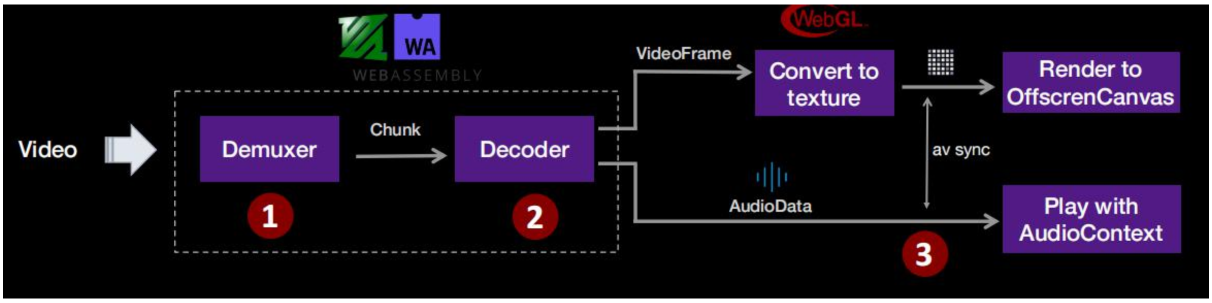
将像素数据转为 WebGL 纹理绘制到 Canvas；
将音频原始采样数据 PCM 发送给 WebContext.AudioWorklet 播放；
根据音频的播放时间来挑选视视频帧并绘制，保证音画同步。

以上步骤中的 1）和 2）如何在 Web 中实现？

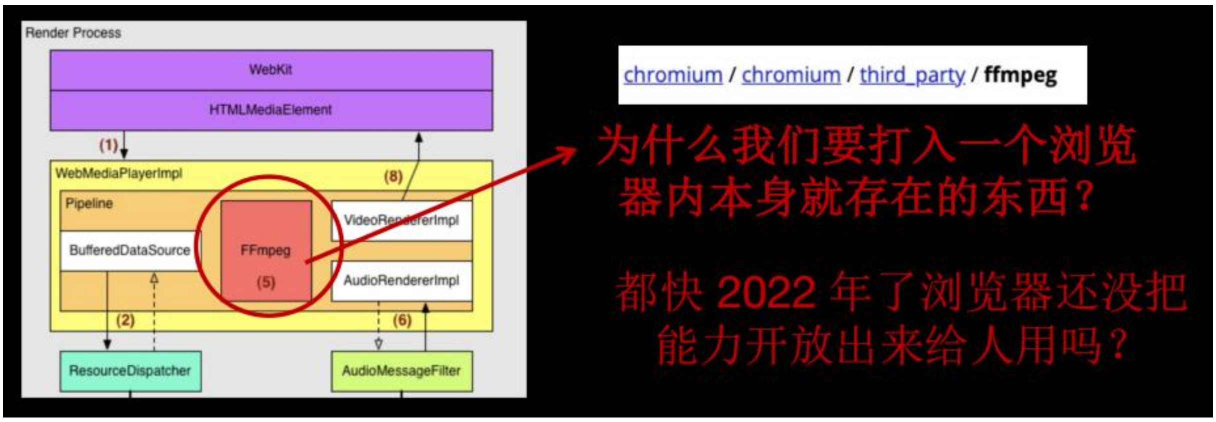
传统的方式是使用 FFmpeg+WebAssembly，步骤如下：

- 根据实际需求对 FFmpeg 进行裁剪定制；
- 编译 FFmpeg 到 WebAssembly；
- 浏览器加载调用；

这个方式固然可行但也存如占用内存空间、加载成本等问题。



在考察浏览器如何解析和播放视频时发现，在 HTMLMediaElement 的架构下使用的就是 FFmpeg，同时 chromium 库里也有 FFmpeg，是否浏览器已经开放这些能力共享呢？



W3C 提供的 WebCodecs API 就是这样一个可以解决 Web 端编解码音视频的工具。



WebCodecs

Editor's Draft, 29 November 2021

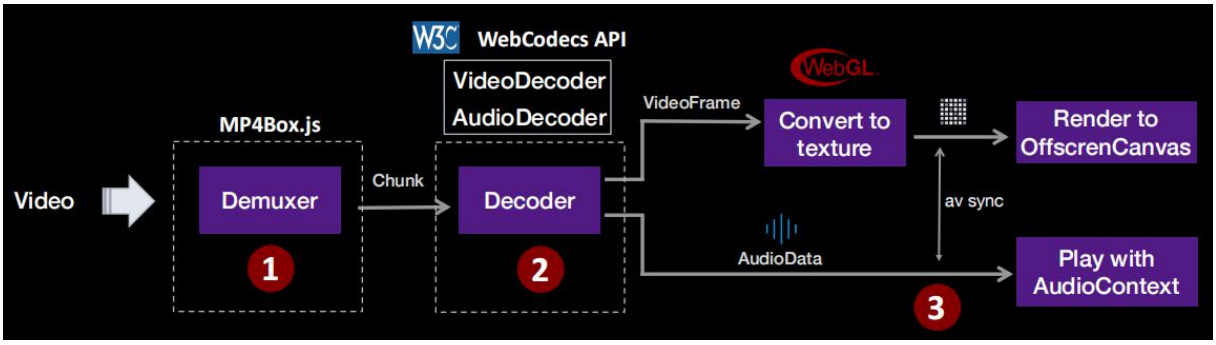
Abstract

This specification defines interfaces to codecs for encoding and decoding of audio, video, and images.

WebCodecs API

The **WebCodecs API** gives web developers low-level access to the individual frames of a video stream and chunks of audio. It is useful for web applications that require full control over the way media is processed. For example, video or audio editors, and video conferencing.

在第二步解码使用 WebCodecs API，其优势在于不但提供了视频音频解码器，还有图像解码器和视频音频编码器。但唯一的问题是 WebCodecs 没有提供相应的解封装 API，这就需要借助 MP4Box.js/mux.js/ffmpeg 之类的工具拿到原始编码数据。

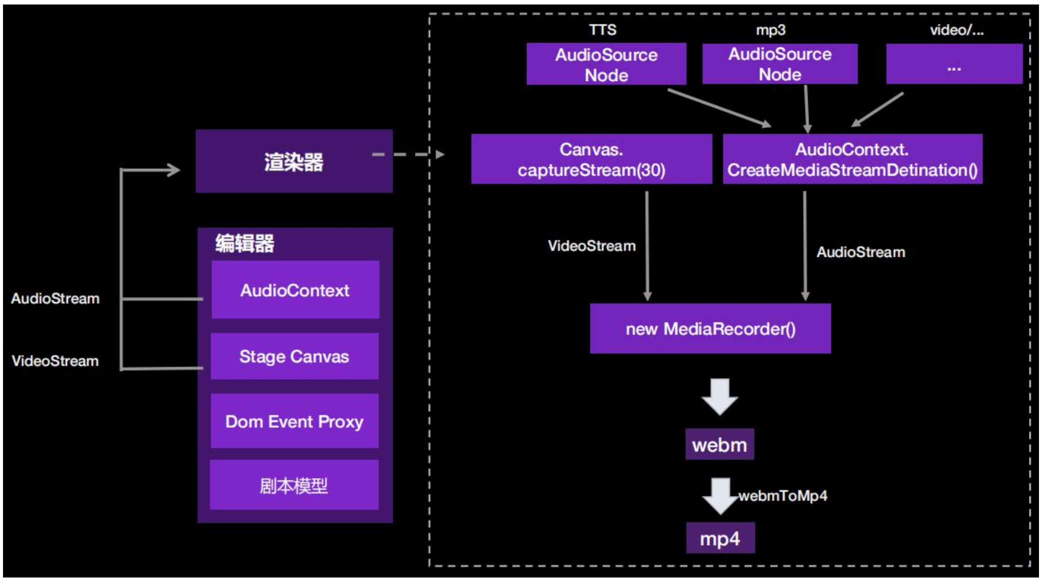


4、渲染出片

渲染包括视频渲染和音频渲染。下图右边部分展示了常规的渲染出片流程。

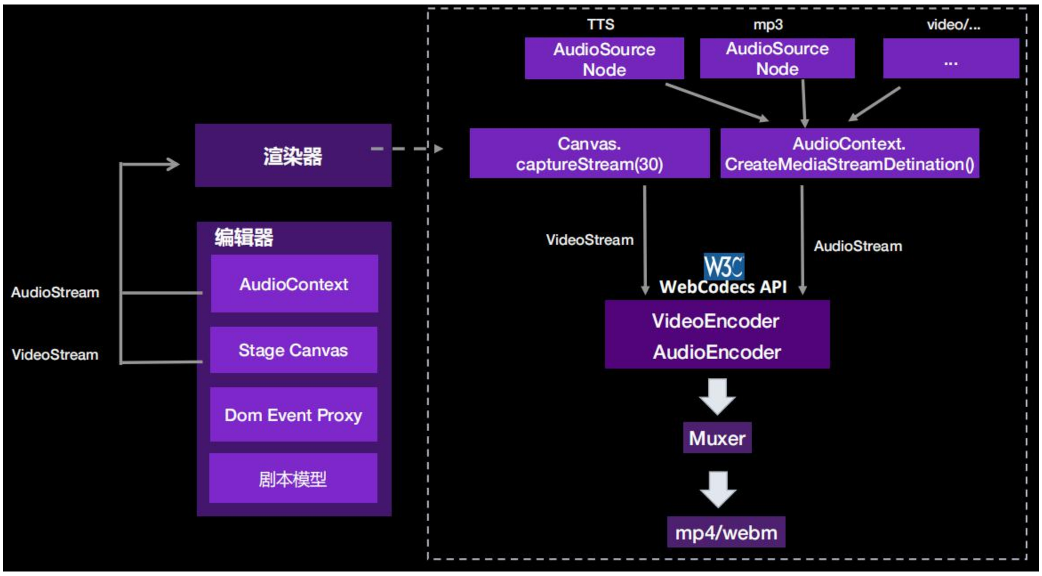
音频渲染可以通过 AudioContext 创建 MediaStreamDestination 节点，将音频文件整合到这个节点上，然后采集所需的 AudioSteam。

视频渲染是在 Canvas 上按一定的帧率捕捉得到 VideoStream，将 VideoStream 和 AudioStream 输入到 MediaRecorder API 中，生成 webm 格式视频，再转换成 mp4 格式。



常规渲染出片流程

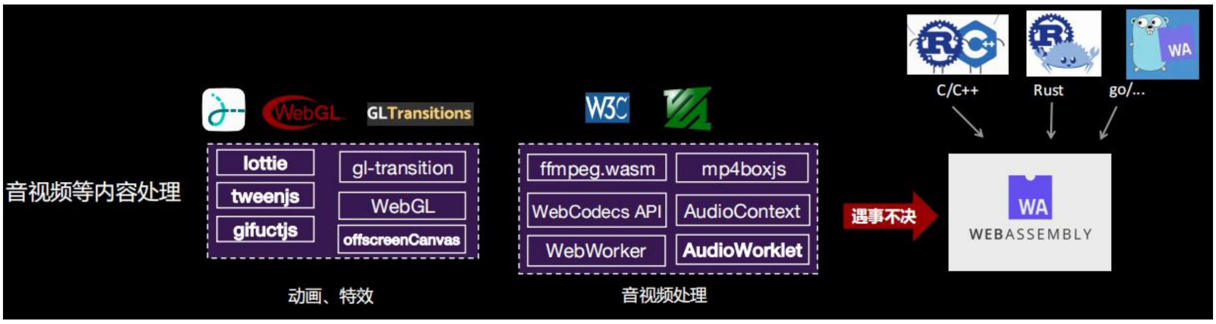
由于 WebCodecs 也同时提供音视频编辑功能，因此可以代替流程中的 new MediaRecorder，编辑解析后经过分装然后，直接输出需要的格式视频。



使用 WebCodecs API 的渲染出片流程

四、浏览器实现的问题与挑战

Web 上已经具备音视频所需的所有能力，包括提供动画、特效能力的 WebGL，提供音视频处理能力的 WebCodecs、FFmpeg，以及 WebAssembly，因此在浏览器实现音视频编辑是可行的。



如果作为专业的音视频编辑器，浏览器实现的问题和挑战有以下方面：

1、文件处理

尽管使用 dexie.js 通过 indexDB 存储但视频文件和素材通常很大，所以目前的方案仅适用于短视频的创作（60s 左右）；

解决方案：可以通过能直接访问系统的 API，如 File System Access API 实现对于大文件的编辑，在未来是可以解决的。

2、长视频的渲染

MediaRecorder 的方式出片需要和视频一样时长的等待，长视频则会消耗用户大量的等待时间。解决方案：将长视频拆分，或采用异步处理端侧渲染，FFmpeg 录制浏览器拆分渲染再合并。

3、更多格式的支持

需要找很多 demuxer、muxer 库，但是在 W3C 没有提供更多可复用 API 的情况下，可以使用 FFmpeg 转存 WebAssembly 的方式。

综上所述，在 Web 端的音视频编辑器是可以实现的，在未来，编辑器会有更多机会和优化空间，可以真正实现像做 PPT 一样做视频。

关注「Alibaba F2E」微信公众号
把握阿里巴巴前端新动向

阿里云开发者“藏经阁”
海量电子手册免费下载